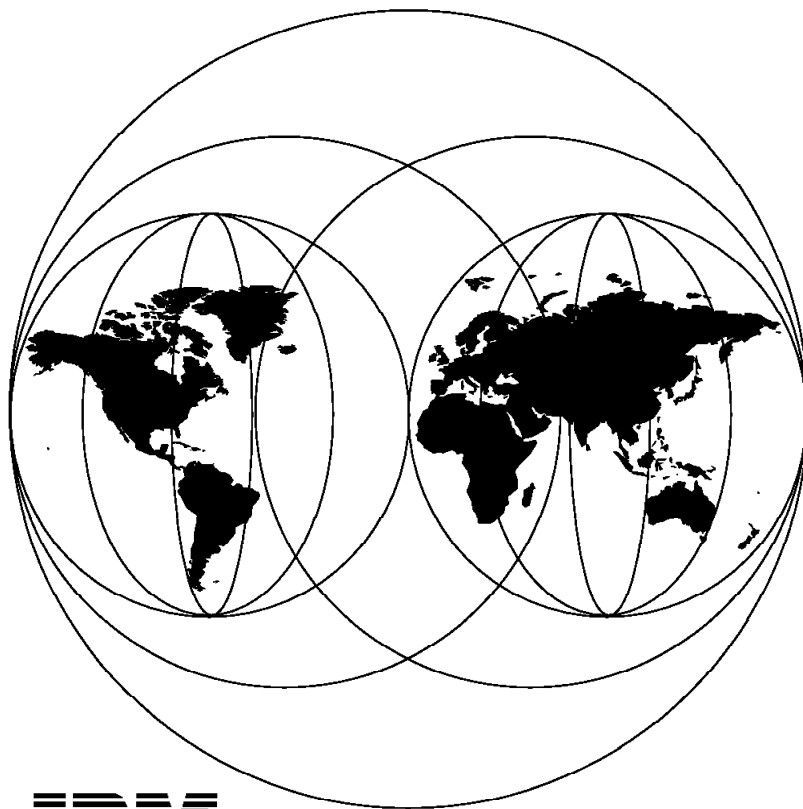


International Technical Support Organization

SG24-4606-00

**IBM VisualAge for COBOL for OS/2  
Object-Oriented Programming**

January 1996



**International Technical Support Organization  
San Jose Center**





International Technical Support Organization

SG24-4606-00

**IBM VisualAge for COBOL for OS/2  
Object-Oriented Programming**

January 1996

**Take Note!**

Before using this information and the product it supports, be sure to read the general information under "Special Notices" on page xiii.

**First Edition (January 1996)**

This edition applies to Version 1, Release 1, of IBM VisualAge for COBOL for OS/2 (Part number 28H2177) for use with OS/2 2.1 or higher. The examples in this edition require a Corrective Service Disk (CSD) that is planned to be applied to IBM VisualAge for COBOL for OS/2 Version 1, Release 1 product in the first quarter of 1996.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

An ITSO Technical Bulletin Evaluation Form for reader's feedback appears facing Chapter 1. If the form has been removed, comments may be addressed to:

IBM Corporation, International Technical Support Organization  
Dept. 471 Building 80-E2  
650 Harry Road  
San Jose, California 95120-6099

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© **Copyright International Business Machines Corporation 1996. All rights reserved.**

Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

---

## Abstract

This document introduces and illustrates the techniques needed to start exploiting the immense possibilities offered by the object-oriented capabilities of IBM's VisualAge for COBOL for OS/2 and related products in the IBM COBOL family of products.

It introduces basic object-oriented concepts, describes in an illustrated manner how to code those concepts using IBM VisualAge for COBOL for OS/2, and provides a fictional account of how an application was written using an object-oriented approach with IBM VisualAge for COBOL for OS/2.

This document was written for anyone expecting to be involved in the production of object-oriented COBOL programs and presumes knowledge of COBOL.

(349 pages)



---

# Contents

<b>Abstract</b>	iii
<b>Special Notices</b>	xiii
<b>Preface</b>	xv
How This Document is Organized	xv
Related Publications	xvii
International Technical Support Organization Publications	xvii
ITSO Redbooks on the World Wide Web (WWW)	xviii
Acknowledgments	xviii
<hr/>	
<b>Part 1. Introducing Object-Oriented COBOL</b>	1
<b>Chapter 1. Introduction</b>	3
1.1 The Importance of OO COBOL	3
1.2 Purpose of Manual	4
1.3 Assumption Concerning Audience	4
1.4 Manual Structure	5
<b>Chapter 2. OO Concepts</b>	7
2.1 The Direction of Computing	7
2.2 IBM's Leading Role	8
2.3 Application Development Problems	9
2.4 OO Claim to Answers	10
2.5 OO Terminology	12
2.6 OO Review	19
2.7 OO History	20
<b>Chapter 3. Old COBOL Approach</b>	25
3.1 OO Image	25
3.2 Encapsulation	26
3.3 Approach Limitations	28
<b>Chapter 4. New COBOL Approach</b>	31
4.1 Requirements	31
4.2 Explicit Definition	31
4.3 Client/Class Program Relationship	32
4.4 Recursion	33
4.5 Summary of Language Changes	34
4.6 ANSI Standards	35
4.7 Class Libraries	43
4.8 Comparison with Other Languages	43
4.9 Further Development	45
<b>Chapter 5. Implementation</b>	47
5.1 Evolution	47
5.2 Dangers and Difficulties	48
5.3 IBM Internal Resources	49
5.4 Timescales	50
5.5 Conclusion	51

<b>Part 2. Coding Object-Oriented COBOL</b>	<b>53</b>
<b>Chapter 6. Overview</b>	<b>55</b>
6.1 Purpose	55
6.2 Format	55
6.3 Business Problem	56
6.4 Code Creation and Processing	56
<b>Chapter 7. Classes</b>	<b>59</b>
7.1 Class Definition Structure	61
7.2 Class Definition Statements	62
<b>Chapter 8. Methods</b>	<b>65</b>
8.1 Method Definition Structure	65
8.1.1 Method Definition Statements	66
<b>Chapter 9. Client</b>	<b>69</b>
9.1 Client Program Example	69
9.2 Client Definition Structure	69
9.3 Client Definition Statements	70
9.4 Object Reference Statement Details	71
9.5 Interacting with Objects	71
<b>Chapter 10. Example One</b>	<b>73</b>
10.1 WineCase Class Example	73
10.2 UserInterface Class Example	77
10.3 Wine Client Example	80
<b>Chapter 11. Subclasses</b>	<b>81</b>
11.1 Business Example	81
11.2 Subclass Definition Statements	83
11.3 Method Definition Statements	84
11.4 Accessing Data in Superclasses	85
<b>Chapter 12. Example Two</b>	<b>87</b>
12.1 Winecase	87
12.2 Newcase Class Example	89
12.3 Oldcase Class Example	90
12.4 UserInterface Class Example	93
12.5 Wine Client Program	95
<b>Chapter 13. Metaclasses</b>	<b>97</b>
13.1 Need for Metaclasses	97
13.2 Definition	97
13.3 Classes and Metaclasses Example	98
13.4 Definition	99
<b>Chapter 14. Example Three</b>	<b>103</b>
14.1 MetaOldCase MetaClass	103
14.2 OldCase Class	104
14.3 Wine Client Program	107
<b>Part 3. Object-Oriented COBOL - An Example</b>	<b>109</b>



<b>Chapter 15. The Wine Store Scenario</b>	111
15.1 Background	111
15.2 Overview of the Business Process	111
<b>Chapter 16. OO Analysis and Design Processes</b>	113
16.1 Methodology	113
16.1.1 Analysis	113
16.1.2 Design	114
16.2 Analyzing the Objects	114
16.2.1 A Use Case of the WineStore Application	114
<b>Chapter 17. Object-Oriented COBOL Implementation</b>	121
17.1 Code Creation Process	121
17.2 COBOL Code Creation Structure	121
17.3 Code Commentary	122
17.3.1 Wine Client	122
17.3.2 WineOrder Class	126
17.3.3 UserInterface Class	131
17.3.4 Bottle Class	133
<b>Chapter 18. The Second Iteration</b>	137
18.1 Code Commentary	137
18.1.1 A New Class	137
18.1.2 Object Interaction Diagram	138
18.1.3 Wine Client	139
18.1.4 WineOrder Class	141
18.1.5 UserInterface Class	148
18.1.6 WineBottle Class	149
18.1.7 FileRW Class	150
<b>Chapter 19. The Third Iteration</b>	153
19.1.1 Object Interaction Diagram	158
19.2 Code Design	159
19.3 Code Commentary	161
19.3.1 Wine Client	161
19.3.2 WineOrder Class	163
19.3.3 OldOrder Class	166
19.3.4 UserInterface Class	168
19.3.5 Bottle Class	171
19.3.6 FileRW Class	172
<b>Chapter 20. The Fourth Iteration</b>	175
20.1.1 Object Interaction Diagram	177
20.2 Code Enhancements	177
20.3 Code Commentary	178
20.3.1 Wine Client	178
20.3.2 OldOrder Class	180
20.3.3 UserInterface Class	181
20.3.4 MetaOldOrder Class	182
<b>Appendix A. Example One Source Code</b>	185
A.1 Example One – UserInterface Class Code	185
A.2 Example One – WineCase Class Code	186
A.3 Example One – Wine Client Class Code	189

<b>Appendix B. Example Two Source Code</b>	191
B.1 Example Two – UserInterface Class Code	191
B.2 Example Two – WineCase Class Code	193
B.3 Example Two – Wine Client Class Code	196
B.4 Example Two – NewCase Class Code	197
B.5 Example Two – OldCase Class Code	197
 <b>Appendix C. Example Three Source Code</b>	 201
C.1 Example Three – UserInterface Call Code	201
C.2 Example Three – WineCase Class Code	203
C.3 Example Three – Wine Client Class Code	206
C.4 Example Three – MetaOldCase Metaclass Code	207
C.5 Example Three – OldCase Class Code	208
C.6 Example Three – NewCase Class Code	210
 <b>Appendix D. Wine Store Scenario – Iteration One Code</b>	 211
D.1 Wine Client Code	211
D.2 Wine Order Class Code	212
D.3 User Interface Class Code	215
D.4 Bottle Class Code	218
 <b>Appendix E. Wine Store Scenario – Iteration Two Code</b>	 221
E.1 Wine Client Code	221
E.2 Wine Order Class Code	225
E.3 User Interface Class Code	235
E.4 Bottle Class Code	240
E.5 FileRW Class Code	244
 <b>Appendix F. Wine Store Scenario – Iteration Three Code</b>	 247
F.1 Wine Client Code	247
F.2 Wine Order Class Code	253
F.3 Old Order Class Code	266
F.4 User Interface Class Code	270
F.5 Bottle Class Code	276
F.6 FileRW Class Code	281
 <b>Appendix G. Wine Store Scenario – Iteration Four Code</b>	 285
G.1 Wine Client Code	285
G.2 Old Order Class Code	291
G.3 User Interface Class Code	295
G.4 Meta Old Order Class Code	302
G.5 FileRW Class Code	305
G.6 Bottle Class Code	308
G.7 Order Class Code	312
 <b>Glossary</b>	 327
 <b>List of Abbreviations</b>	 347
 <b>Index</b>	 349

## Figures

1.	The Importance of OO COBOL	4
2.	Manual Structure	5
3.	The Direction of Computing	7
4.	IBM and OO	8
5.	AD New Technology	9
6.	The Promise of OO	11
7.	OO Terminology	12
8.	OO Terminology	13
9.	Encapsulation	14
10.	Inheritance	15
11.	Polymorphism	16
12.	Class-Object Relationship	17
13.	Messages	18
14.	OO Terminology	19
15.	How OO Scores	20
16.	OO History	21
17.	OO COBOL Standards	22
18.	Object Standardization	23
19.	OO Image versus Reality	25
20.	Encapsulation	26
21.	Inheritance	27
22.	Polymorphism	28
23.	Limitations	29
24.	OO Requirements	31
25.	Explicitness	32
26.	Client/Class Relationship	33
27.	Recursion	34
28.	Syntax	35
29.	ANSI Standards	36
30.	Problems without SOM	37
31.	SOM Definition	38
32.	How SOM Works	39
33.	Platform Support	40
34.	Som and DSOM	41
35.	Programmer Use of SOM	42
36.	Implications	43
37.	OO Languages	44
38.	Comparison of OO languages	45
39.	Further Requirements	46
40.	Implementation	47
41.	Dangers and Difficulties	48
42.	Education	49
43.	Internal IBM Resources	50
44.	Timescale	51
45.	Breakthrough	52
46.	CRC Card for WineOrder Class	118
47.	CRC Card for WineBottle Class	119
48.	CRC Card for UserInterface Class	120
49.	CRC Card for Wine Store Scenario Second Iteration	138
50.	Object Interaction Diagram – Second Iteration	138
51.	CRC Card for WineOrder Class in Third Iteration	155

52.	CRC Card for OldOrder Class in Third Iteration . . . . .	156
53.	CRC Card for WineBottle Class in Third Iteration . . . . .	157
54.	CRC Card for FileRW Class in Third Iteration . . . . .	157
55.	CRC Card for UserInterface Class in Third Iteration . . . . .	158
56.	Object Interaction Diagram – Third Iteration . . . . .	159
57.	CRC Card for MetaOldOrder in Fourth Iteration . . . . .	176
58.	Object Interaction Diagram – Fourth Iteration . . . . .	177

---

## Tables

1.	Winecase Class with Data and Methods	60
2.	Userint Class with Data and Methods	60
3.	Winecase Class with Data and Methods – Review	65
4.	Userint Class with Data and Methods – Review	65
5.	Client Program Logic Flow Part One	69
6.	Client Program Logic Flow Part Two	69
7.	Client Program Logic Flow Part Three	69
8.	Winecase Class with Data and Methods (Subclasses)	81
9.	Data Class (Subclasses)	82
10.	Newcase Class (Subclasses)	82
11.	Userint Class (Subclasses)	82
12.	Winecase Superclass Data and Methods	85
13.	Classes and Metaclasses	98
14.	WineOrder Class Attributes	117
15.	WineOrder Class Methods	117
16.	WineBottle Class Attributes	118
17.	WineBottle Class Methods	118
18.	UserInterface Class Attributes	119
19.	UserInterface Class Methods	119
20.	Wine Store Scenario Methods for Second Iteration	137
21.	WineOrder Class Attributes for Third Iteration	154
22.	WineOrder Class Methods for Third Iteration	154
23.	OldOrder Class Attributes for Third Iteration	155
24.	OldOrder Class Methods for Third Iteration	155
25.	WineBottle Class Attributes for Third Iteration	156
26.	WineBottle Class Methods for Third Iteration	156
27.	FileRW Class Methods for Third Iteration	157
28.	UserInterface Class Attributes for Third Iteration	157
29.	UserInterface Class Methods for Third Iteration	158
30.	MetaOldOrder Class Attribute for Fourth Iteration	176
31.	MetaOldOrder Class Methods for Fourth Iteration	176
32.	UserInterface Class Methods for Fourth Iteration	176



---

## Special Notices

This publication is intended to help customers to implement object-oriented systems and applications using products in the IBM COBOL family of products. The information in this publication is not intended as the specification of any programming interfaces that are provided by IBM COBOL for MVS and VM, by IBM VisualAge for COBOL for OS/2, or by IBM COBOL Set for AIX. See the PUBLICATIONS section of the IBM Programming Announcement for IBM COBOL for MVS and VM, IBM VisualAge for COBOL for OS/2, and IBM COBOL Set for AIX for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594 USA.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

IBM

The following terms are trademarks of other companies:

C-bus is a trademark of Corollary, Inc.

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Windows is a trademark of Microsoft Corporation.

Other trademarks are trademarks of their respective companies.



---

## Preface

This document is intended to complement the IBM COBOL family manuals by providing an example of object-oriented development.

This document is intended to be of value to anyone involved in object-oriented development using IBM COBOL on MVS, AIX, or OS/2, although the example itself runs on OS/2.

---

## How This Document is Organized

The document is organized as follows:

- Chapter 1, "Introduction"

The importance of OO COBOL and the structure of this manual are presented this chapter.

- Chapter 2, "OO Concepts"

This provides the basic OO awareness and terminology and provides a basis for the remainder of this document.

- Chapter 3, "Old COBOL Approach"

This chapter describes how old COBOL can provide elements of object orientation, looking at the limitations of this approach.

- Chapter 4, "New COBOL Approach"

This chapter describes the new object-oriented facilities provided in the IBM COBOL product family to support OO computing, in particular looking at the role of SOM.

- Chapter 5, "Implementation"

This chapter describes how OO COBOL systems might be introduced and concludes the introduction.

- Chapter 6, "Overview"

This chapter introduces coding syntax of object-oriented COBOL for the IBM product family using IBM VisualAge for COBOL for OS/2.

- Chapter 7, "Classes"

This chapter explains the syntax needed for classes.

- Chapter 8, "Methods"

This chapter explains the syntax needed for methods.

- Chapter 9, "Client"

This chapter shows how to code client programs.

- Chapter 10, "Example One"

This chapter presents an example of the syntax explained thus far.

- Chapter 11, "Subclasses"

This chapter describes the coding of subclasses.

- Chapter 12, “Example Two”  
The example in this chapter shows how subclasses can be added to the previous example.
- Chapter 13, “Metaclasses”  
This chapter introduces the concepts and syntax of metaclasses.
- Chapter 14, “Example Three”  
This chapter adds metaclasses to our previous example.
- Chapter 15, “The Wine Store Scenario”  
This chapter introduces the business example application, the Wine Store Scenario.
- Chapter 16, “OO Analysis and Design Processes”  
This chapter describes the object-oriented analysis and design involved in the initial part of the development for the Wine Store Scenario.
- Chapter 17, “Object-Oriented COBOL Implementation”  
This chapter describes the coding for the first iteration for the Wine Store Scenario.
- Chapter 18, “The Second Iteration”  
This chapter describes the coding for the second iteration for the Wine Store Scenario.
- Chapter 19, “The Third Iteration”  
This chapter describes the coding for the third iteration for the Wine Store Scenario.
- Chapter 20, “The Fourth Iteration”  
This chapter describes the coding for the fourth iteration for the Wine Store Scenario.
- Appendix A, “Example One Source Code”  
This appendix lists the source code for example one.
- Appendix B, “Example Two Source Code”  
This appendix lists the source code for example two.
- Appendix C, “Example Three Source Code”  
This appendix lists the source code for example three.
- Appendix D, “Wine Store Scenario – Iteration One Code”  
This appendix lists the source code for the Wine Store example in its first iteration.
- Appendix E, “Wine Store Scenario – Iteration Two Code”  
This appendix lists the source code for the Wine Store example in its second iteration.
- Appendix F, “Wine Store Scenario – Iteration Three Code”  
This appendix lists the source code for the Wine Store example in its third iteration.

- Appendix G, “Wine Store Scenario – Iteration Four Code”

This appendix lists the source code for the Wine Store example in its fourth iteration.

---

## Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this document.

- *IBM COBOL for MVS and VM - Programming Guide*, SC26-4767
- *SOMobjects for MVS - User's Guide*, GC28-1545
- *SOMobjects for MVS - Class Library Reference*, SC28-1546
- *IBM COBOL Set for AIX - Programming Guide*, SC26-8423
- *IBM COBOL Set for AIX - Getting Started*, GC26-8425
- *IBM VisualAge for COBOL for OS/2 - Programming Guide*, SC26-8419
- *IBM VisualAge for COBOL for OS/2 - Getting Started*, GC26-8421

---

## International Technical Support Organization Publications

A complete list of International Technical Support Organization publications, known as redbooks, with a brief description of each, may be found in:

*International Technical Support Organization Bibliography of Redbooks*, GG24-3070.

To get a catalog of ITSO redbooks, VNET users may type:

```
TOOLS SENDTO WTSCPOK TOOLS REDBOOKS GET REDBOOKS CATALOG
```

A listing of all redbooks, sorted by category, may also be found on MKTTOOLS as ITSOCAT TXT. This package is updated monthly.

---

### How to Order ITSO Redbooks

IBM employees in the USA may order ITSO books and CD-ROMs using PUBORDER. Customers in the USA may order by calling 1-800-879-2755 or by faxing 1-800-445-9269. Most major credit cards are accepted. Outside the USA, customers should contact their local IBM office. For guidance on ordering, send a note to BOOKSHOP at DKIBMVM1 or E-mail to [bookshop@dk.ibm.com](mailto:bookshop@dk.ibm.com).

Customers may order hardcopy ITSO books individually or in customized sets, called BOFs, which relate to specific functions of interest. IBM employees and customers may also order ITSO books in online format on CD-ROM collections, which contain redbooks on a variety of products.

---

## ITSO Redbooks on the World Wide Web (WWW)

Internet users may find information about redbooks on the ITSO World Wide Web home page. To access the ITSO Web pages, point your Web browser to the following URL:

<http://www.redbooks.ibm.com/redbooks>

IBM employees may access LIST3820s of redbooks as well. The internal Redbooks home page may be found at the following URL:

<http://w3.itsc.pok.ibm.com/redbooks/redbooks.html>

---

## Acknowledgments

This project was designed and managed by:

Joe DeCarlo  
International Technical Support Organization, San Jose Center

The authors of this document are:

Richard Mascal  
IBM United Kingdom

Rob Pittman  
IBM North America

This publication is the result of a residency conducted at the International Technical Support Organization, San Jose Center.

Thanks to the following people for the invaluable advice and guidance provided in the production of this document:

George Forshay  
Services Development, Santa Teresa Laboratory, IBM Software Solutions

Ira Sheftman  
COBOL Development, Santa Teresa Laboratory, IBM Software Solutions

---

## Part 1. Introducing Object-Oriented COBOL



---

## Chapter 1. Introduction

This chapter explains the present and future of object-oriented programming and COBOL.

---

### 1.1 The Importance of OO COBOL

Looking at the history of commercial computing, one subject stands out in its importance and pervasiveness: COBOL. Even today, when computer journalists, industry watchers, and academics examine mainframe computing and traditional systems, COBOL is still running on 70+% of the world's top organizations' computers. In addition, there are an estimated 70 billion lines of code and over 5 million active COBOL programmers.

If COBOL has dominated the past, and still dominates the present, what will dominate the future? Many commentators point to object-oriented programming (OO) as the direction of the industry. By "industry" we mean all computing: from the largest mainframe, to workstations and the smallest personal computing accessories through tiny real-time devices not conventionally thought of as computers.

Does this mean COBOL is dying and that its usefulness is coming to an end?

On the contrary, IBM's latest enhancements to the COBOL family will move OO from the highly promising to the widely utilized.

The significance for IBM is that whichever company can provide that significant combination of COBOL and OO on all computer platforms will be assured of an immensely powerful and successful position. This is illustrated in Figure 1.

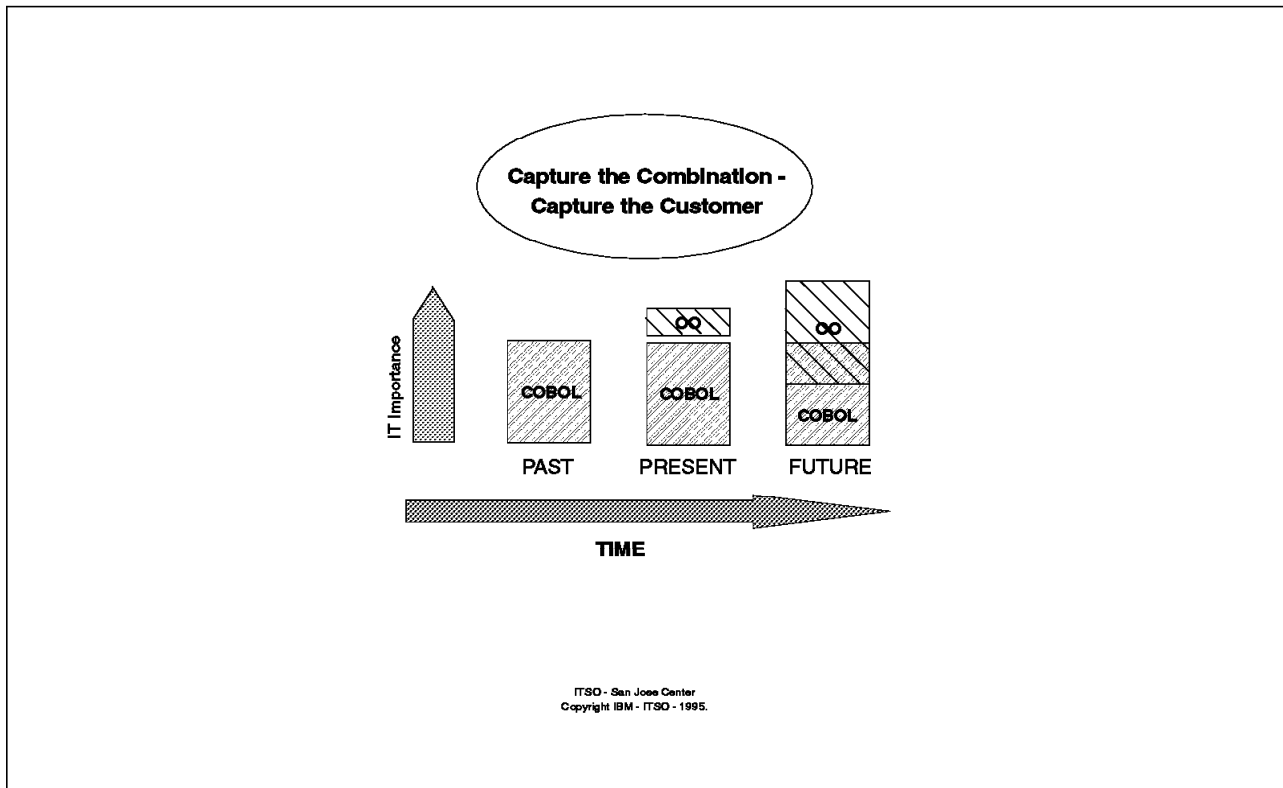


Figure 1. The Importance of OO COBOL

## 1.2 Purpose of Manual

Part one of this manual is designed to communicate the WHAT rather than the HOW of OO COBOL. It will prepare you to discuss the salient features of OO COBOL distinguishing the genuinely new over the already possible and stressing the extra benefits of adopting the IBM approach.

The syntax of the new language facilities are not explained in Part one but is covered in Parts two and three. However, you will be in an excellent position to move on to learning the HOW using this manual as a solid base.

## 1.3 Assumption Concerning Audience

Since no COBOL syntax is mentioned and no knowledge of COBOL is assumed, it is taken for granted that you will understand how COBOL fits in to the traditional transaction management/database commercial systems of today.

In addition, no real familiarity with OO concepts is assumed. Most people have more than that; however this varies widely and whatever level was assumed would mean a mismatch for some readers. Be patient until we reach your level of understanding.



## 1.4 Manual Structure

The topics in Part one are in the sequence shown in Figure 2. This shows a logical progression for 6 definitions of OO, through to how to implement much of OO in today's COBOL. The text shows the difficulties and handicaps of this approach to explain what the new COBOL offers us. Parts two and three illustrate the new OO COBOL syntax using sample applications.

The new COBOL topic forms the heart of this manual and it contains examples using IBM's System Object Model (SOM), which is a major differentiator of IBM's OO approach from others.

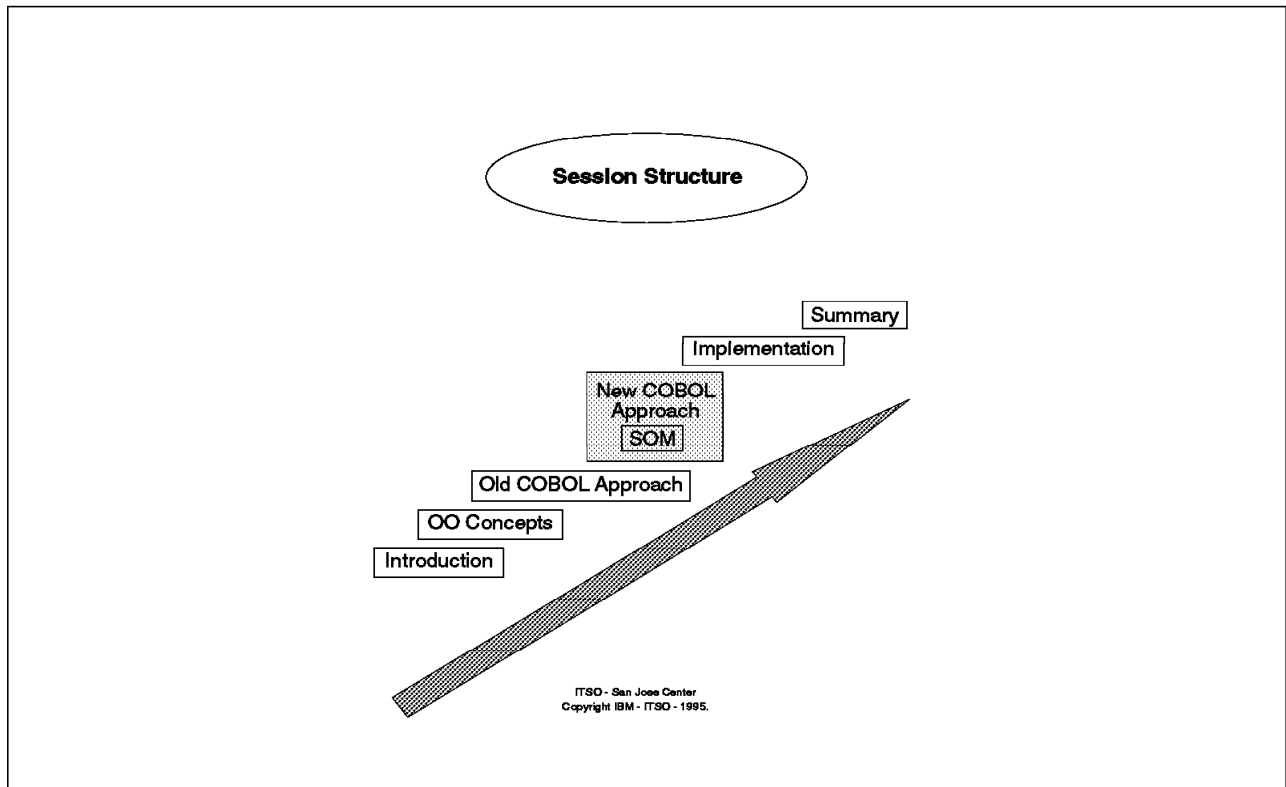


Figure 2. Manual Structure



---

## Chapter 2. OO Concepts

This chapter explains basic object-oriented programming (OO) concepts.

---

### 2.1 The Direction of Computing

Cliff Reeves, IBM's former Director of Object Technology, described a vision of computing in the future as "solely" objects talking to other objects as shown in Figure 3. From the largest machines to machines so small we might not even describe them as computers, from today's machines through to boxes unpredicted even by today's fertile imaginations, they will all be talking to themselves and, potentially, to each other.

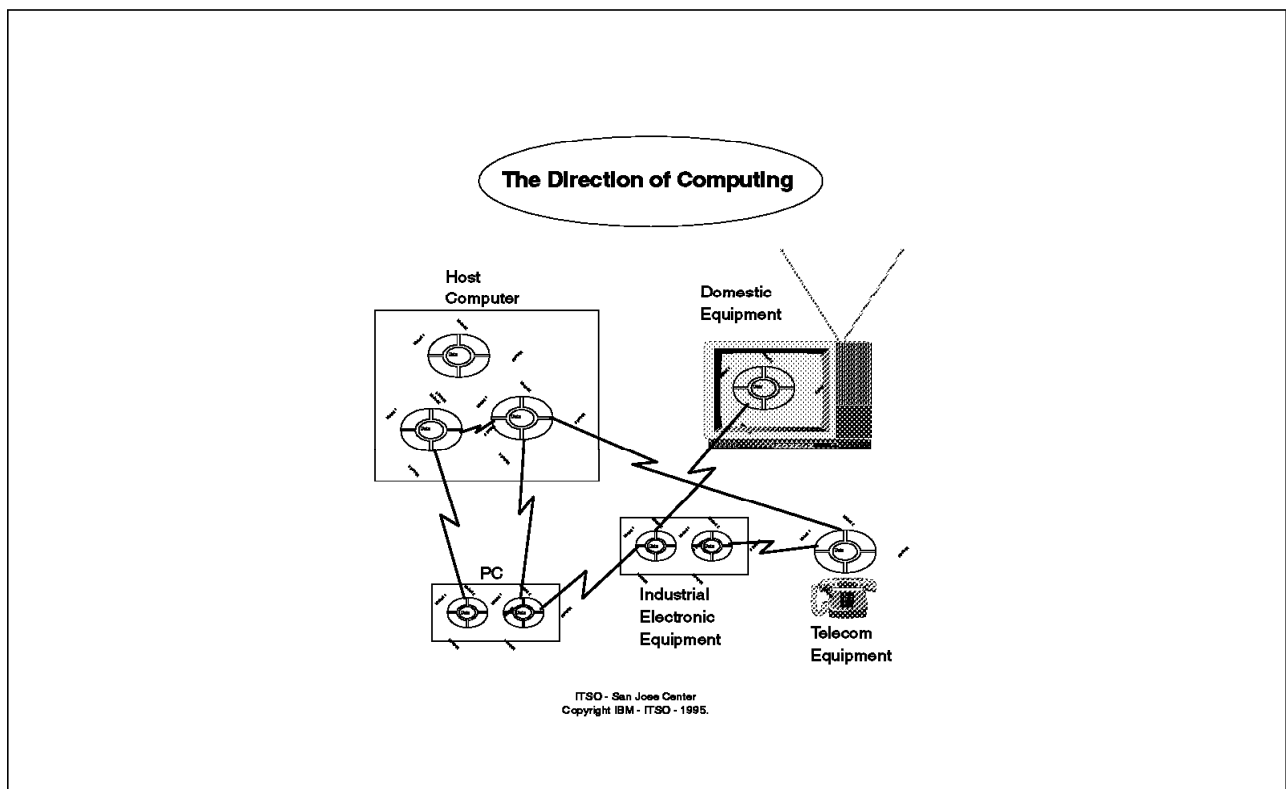


Figure 3. The Direction of Computing

The unit of computing is the object. This is the atomic component, indivisible by other computing constructs. The object is written and plugged together in larger units, as atoms are grouped into molecules and thence to everything we see around us.

There are many reasons why this might be more than a wild prediction by someone with a vested interest. For the moment we can consider just two: the engineering analogy and the client-server paradigm.

The engineering analogy says that if ever the production of software is going to catch up with demand, it will need to adopt the engineering approach of assembly of standardized parts. This is not a novel proposal but what is new is the prospect of object technology fulfilling that role.

The client-server paradigm suggests that computing increasingly separates processing from data, as shown by the use of mainframes as servers, the Internet, and mobile computing. An object-oriented approach is the logical conclusion of a client-server way of system design.

## 2.2 IBM's Leading Role

Object-oriented computing is also fundamental to IBM's vision as shown in Figure 4. Lou Gerstner has cited our object-oriented strategy as one of his most significant focus areas for IBM.

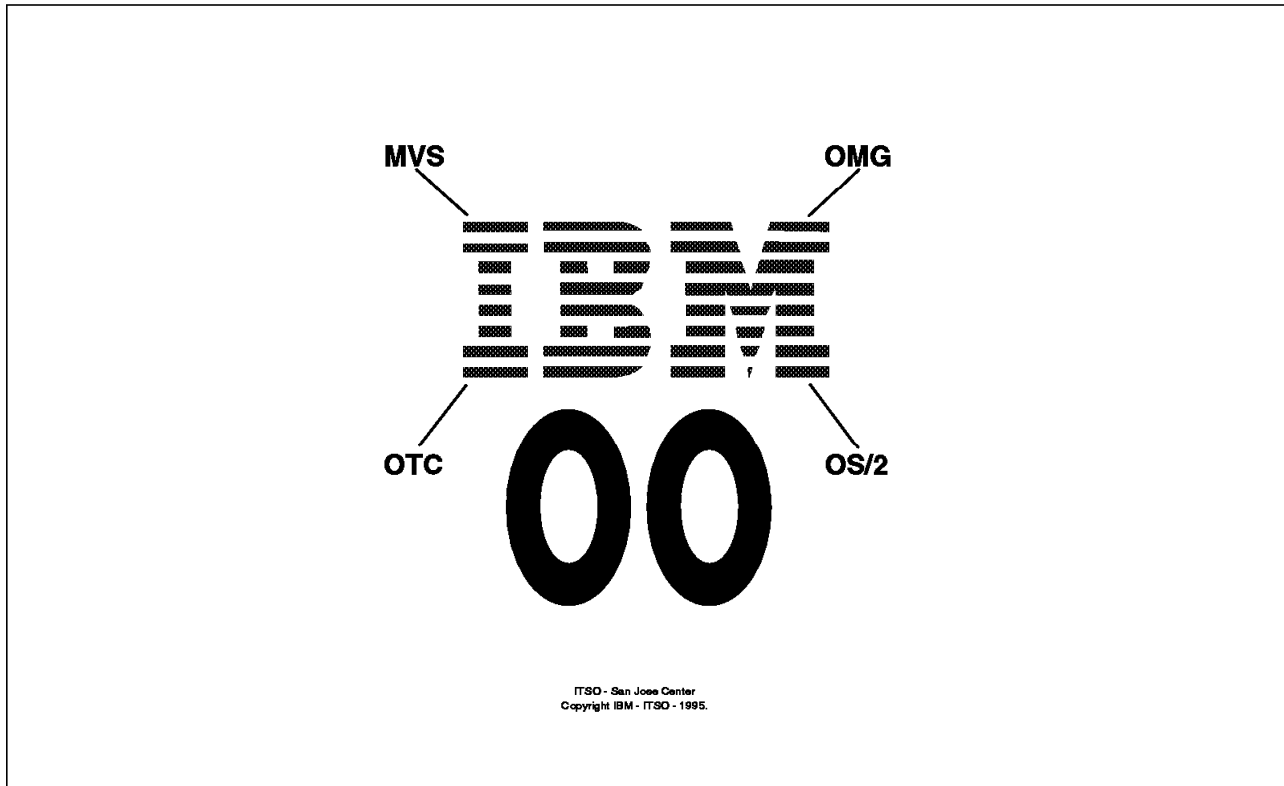


Figure 4. IBM and OO

IBM is involved in such object-oriented activities as its internal group, the OTC (Object Technology Council). OTC promotes the use of OO throughout IBM's development work. An example of its success is the 500,000 lines of MVS products already produced using an object-oriented approach. Much of the visual part of VisualGen was written in Smalltalk and there are many other examples.

Externally too, IBM is fully involved, in particular with the Object Management Group (OMG). This consists of over 400 companies striving to promote the standardized approach required to deliver the greatest benefits of object-oriented technology.

## 2.3 Application Development Problems

Object-oriented technology would not be attracting so much attention had Application Development (AD) been more successful. In fact, AD has been a disaster area. Every country in the industrialized world could come up with example after example of highly visible AD failures such as the baggage-handling system at Denver Airport in the US or the London Ambulance fiasco in the United Kingdom.

For every well publicized catastrophe, there must be thousands of unseen projects overrunning or abandoned. Every survey, formal or informal, of computing customers reveals the same story: AD departments cannot deliver fast enough, what they do produce is not reliable enough, and they do not provide what the users really want.

The near-universality of these complaints shows that it is not simply a matter of incompetence. Could it be the technology employed? This logic was followed back in the mid-50s and led to the introduction in 1957 of the first 3GL, Fortran. Other forward steps are shown in Figure 5.

There have always been problems when new technology has been employed to improve the performance of AD.

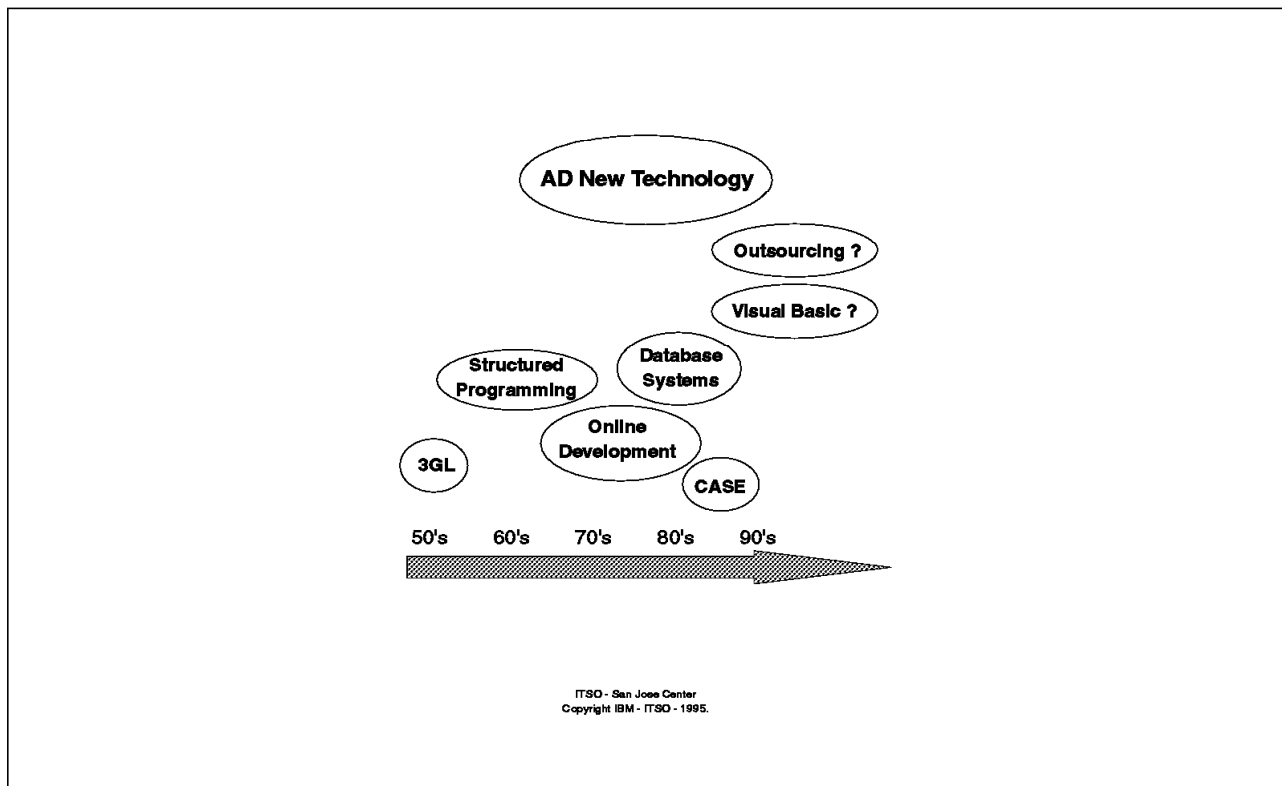


Figure 5. AD New Technology

There has been a succession of advances in our AD capability, often hyped to deliver sooner than they actually did, but nevertheless we are actually building on a history of success. Though that the demands on us grow faster than our ability to improve our effectiveness, we only have to look at the increasing diversity of platforms to see that we need another breakthrough.

Apart from OO, what is there? One approach is outsourcing. Widely tipped as a growth area in the late 80s, actual experience has demonstrated that outsourcing moves the problem around rather than solves it.

Another approach has been to offload the problem to users themselves, or at least their departments. The growth in visual development systems with BASIC in part has been fostered by the resulting need for a simple access to computing power. Simplicity though has its drawbacks as well as its attractions and increasingly surveys are showing that previously hidden costs as they are revealed mean expensive and low industrial strength systems.

---

## 2.4 OO Claim to Answers

Why then do object-oriented supporters claim that this technology promises to be the next breakthrough for AD?

Everyone agrees that greater re-use of code would mean both faster and more productive development as well as the delivery of more reliable code. The difficulty has been in facilitating re-use.

Objects are clearly defined, indivisible combinations of function and data. This suggests that it should prove possible to select objects which have been written earlier. AT&T claims that only 20% of their code now has to be newly written. The rest is reused as a result of using an object-oriented approach. The concept of inheritance further facilitates this suitability for re-use by making simple, local, adjustments to an object possible without copying it.

Users can also relate to descriptions based on objects in a way that they might not with DFDs (Data Flow Diagrams) or ERs (Entity-Relationship) diagrams as shown in Figure 6. like. This should improve the communications between Information Technology (IT) and the user departments.

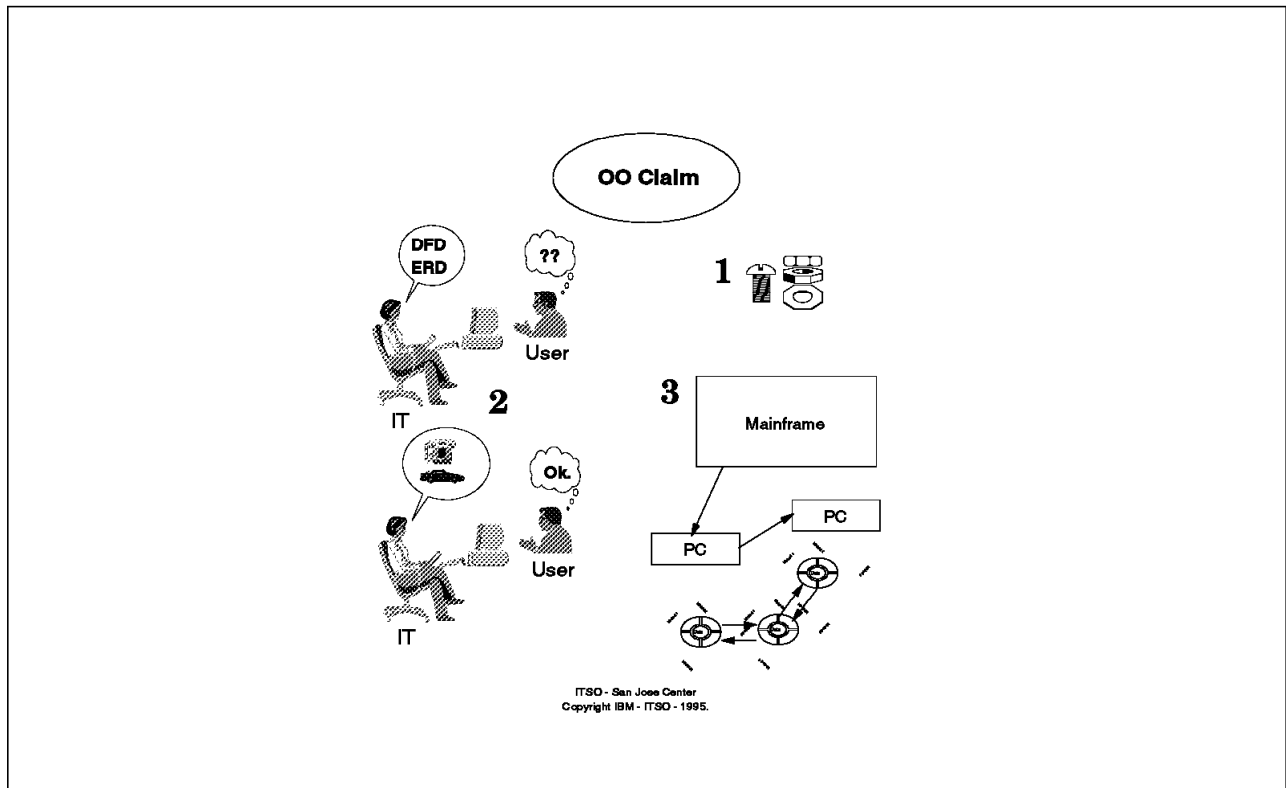


Figure 6. The Promise of OO

Client-server systems might not have exploded in the way predicted but it is plausible to assume that their use will increase. Their mode of operation can be characterized as one unit of processing sending a request to another unit which in turn responds. This picture matches the way objects send messages to each other. This means that the object-oriented approach fits the natural topology of the system architecture.

If object-oriented is such a good idea, why has it not been thought of earlier? It was thought of earlier but there were physical inhibitors which hindered its adoption.

One inhibitor was that the first object-oriented languages required the learning of a completely new system. This inhibitor began to disappear, at least for C programmers, with the introduction of C++.

Another inhibitor was that all technology was either UNIX- or PC-based. This ruled out major commercial systems being created to use object-oriented technology. Finally, since one of the attractions of object-oriented technology is its encouragement of re-use, in particular via the supply of pre-written objects, time is needed to build up a library of these objects. Who wants to invest in producing pre-written objects for sale?

These inhibitors however have begun to diminish in the last few years. Object-oriented programming has moved out of the academic laboratory into the commercial mainstream.

## 2.5 OO Terminology

This section explains the basic terms involved with object-oriented programming. Fortunately, you can say the basics are as easy as PIE (Polymorphism, Inheritance, and Encapsulation) as shown in Figure 7. It is important to first understand objects themselves.

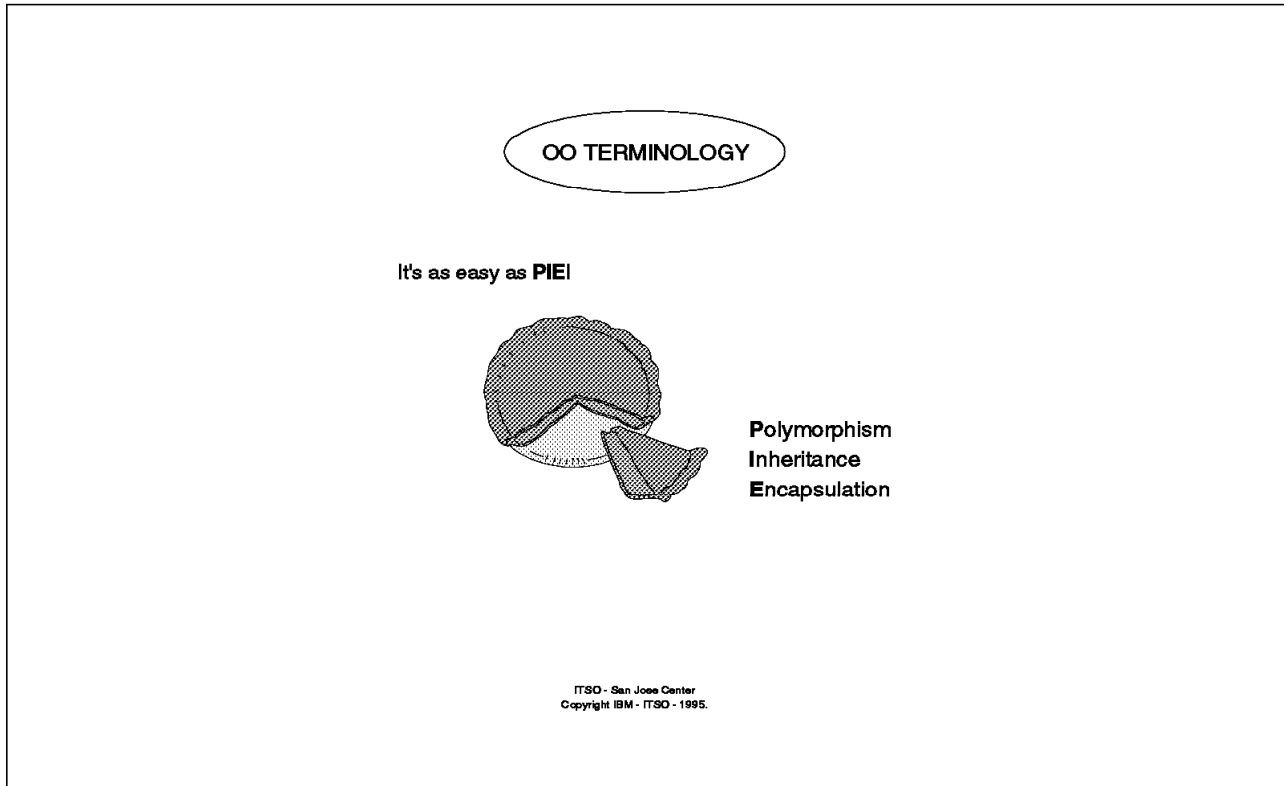


Figure 7. OO Terminology

Objects can represent anything as shown in Figure 8. They might be real tangible things like customers, employees, airplanes, CDs, paintings. They might be less tangible things like accounts, salaries, descriptions. Or they might be system related things like list boxes.



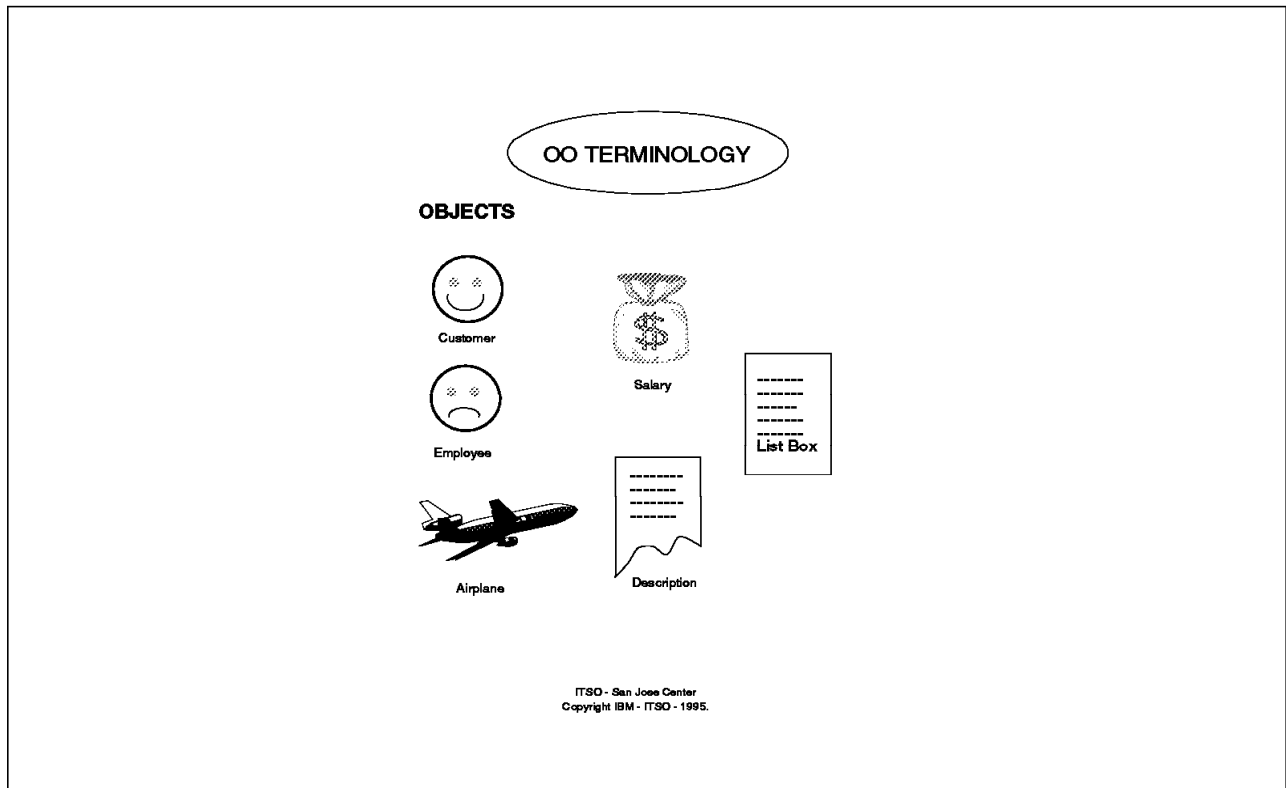


Figure 8. OO Terminology

Objects consist of two sets of components: their data and the functions that work on that data. These functions (called "methods") are the only way of accessing that data, that is, creating, reading, writing. This is called "encapsulation." It is one of the mechanisms which give object-oriented its robustness since it is a strong encourager of data integrity. Encapsulation is illustrated in Figure 9.

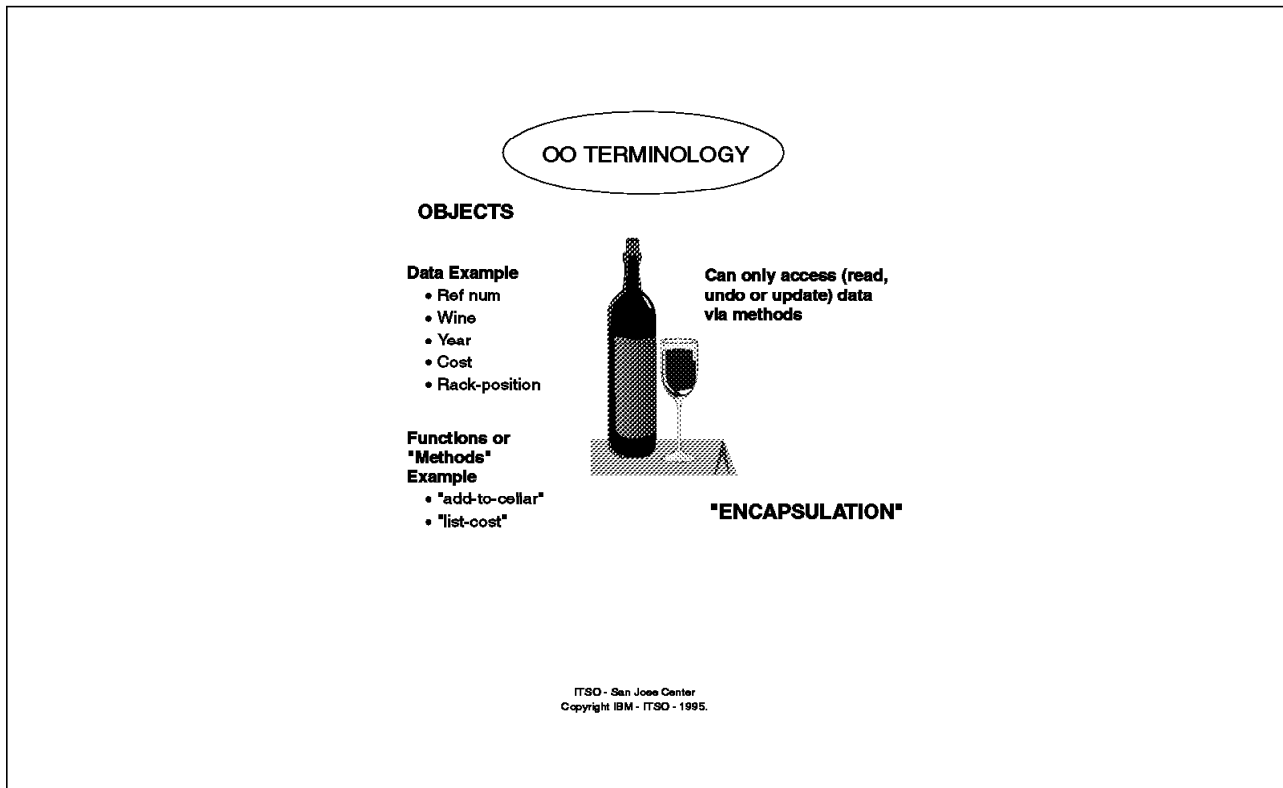


Figure 9. Encapsulation

Often objects are similar in most respects without being identical. Perhaps object B has some more or some less data defined than object A, or one or two extra or fewer methods.

Rather than force the creation of a completely separately defined object, object-oriented programming has the concept of "inheritance" whereby one can define one object as the same as the other but with the different data and/or methods.

Figure 10 shows an example of inheritance. In this example an object "wine-bottle" might have data attributes (name, country, cost, rack-position). To treat your claret differently with a view to its investment value add the extra attribute of "value" and associated methods of "update-value" and "list-value".

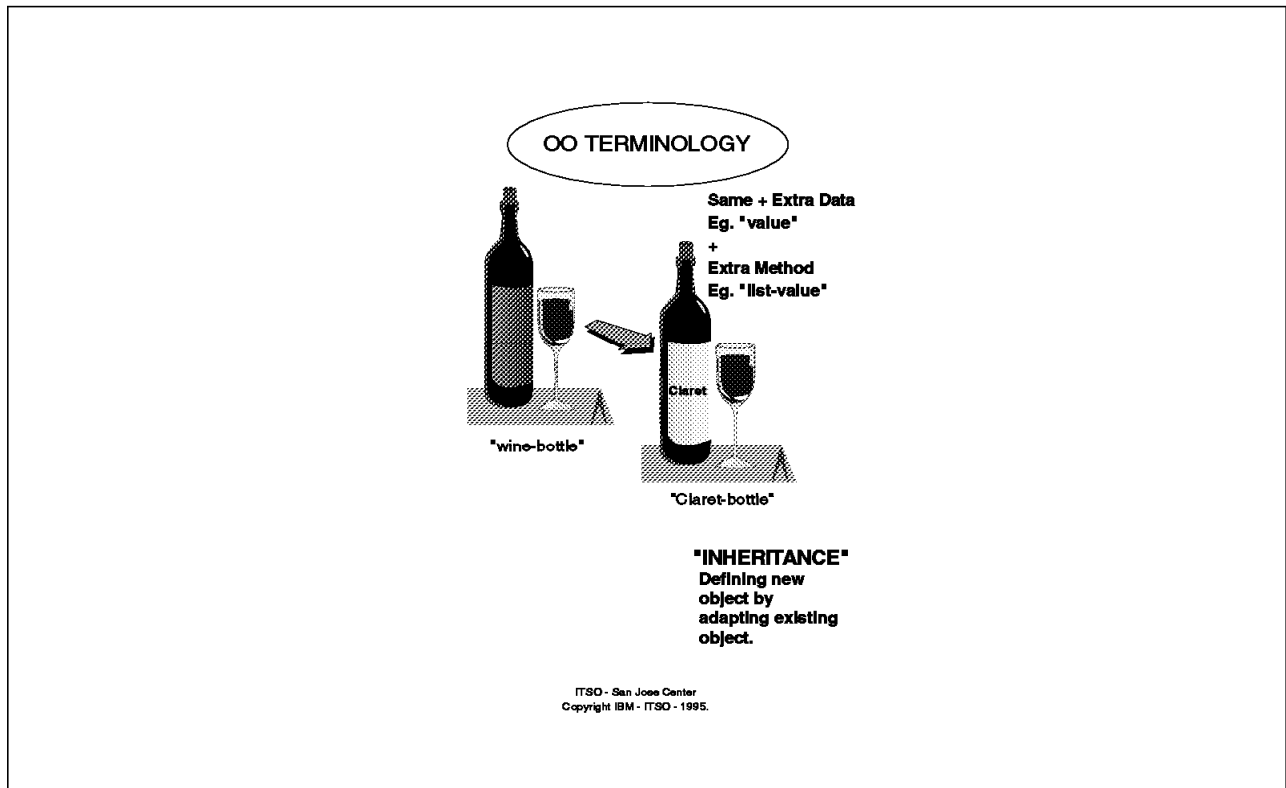


Figure 10. Inheritance

Often it might be that we want the same method to do different things on different objects. In the wine bottle example, we might want the method we invoke when we put a bottle in our cellar, "add-to-cellar", to ask us for name, country, cost normally. If it's a claret bottle we might want it additionally to ask us for a current-value.

This is called "polymorphism," as shown in Figure 11. It makes plugging objects together in a system much easier because we can use the same methods in different circumstances and the system itself takes care of the differences.

The joke is "polymorphism means different things to different people." Unfortunately you need to know what polymorphism does mean in order to understand it.

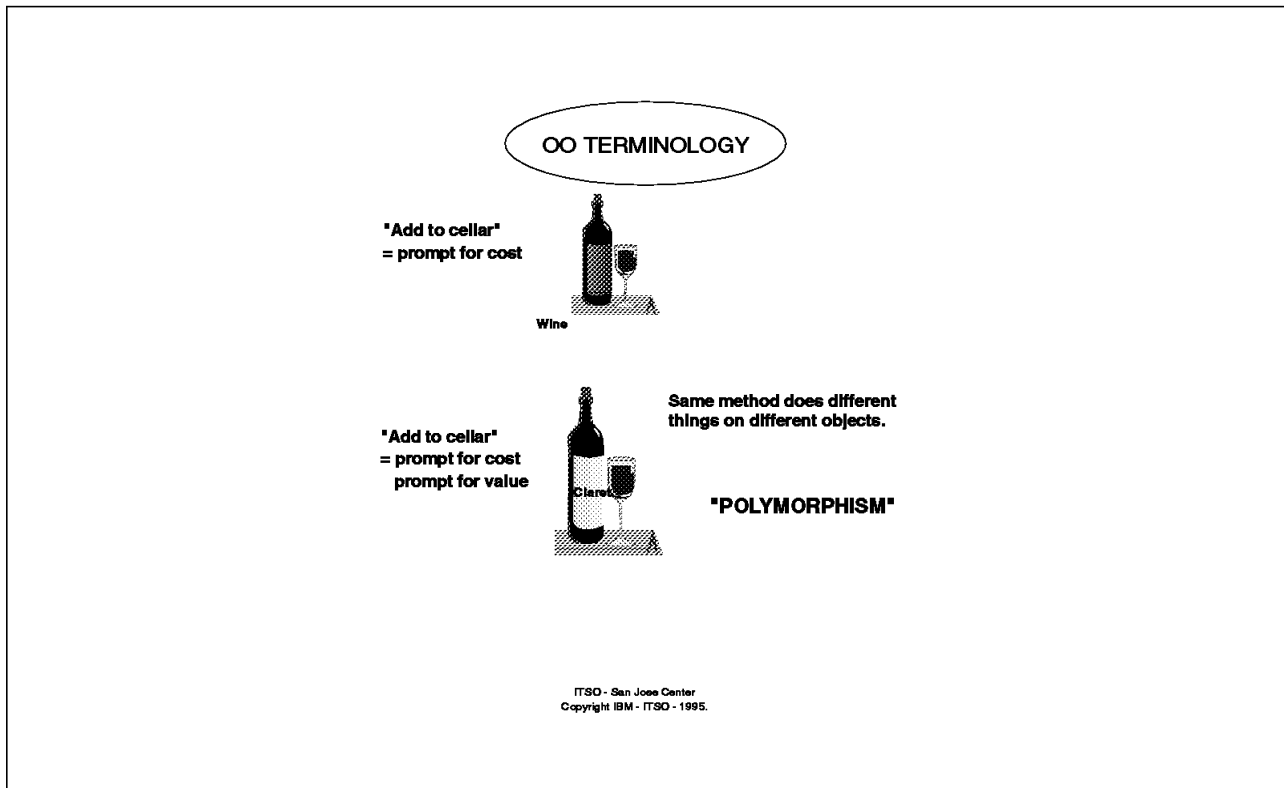


Figure 11. Polymorphism

It is important to distinguish an object from its definition. For example, in the wine-cellar example, there are as many "wine-bottle" objects as there are actual wine bottles in the cellar. But there is a definition of those objects. It says that each one has the data attributes name, country and so forth and that there are certain methods which can work on that data ("add-to-cellar," "list-cost," and so forth).

This definition, or template, as it's often called, is known as a "class." This is illustrated in Figure 12. There is a hierarchy of classes, each class inheriting from another. Each class therefore is a sub-Class, or a child class, of a parent Class.

Another term used to express the relationship of an object to its class is an "instance" of that class. When we create an object we call it "instantiating" the class.

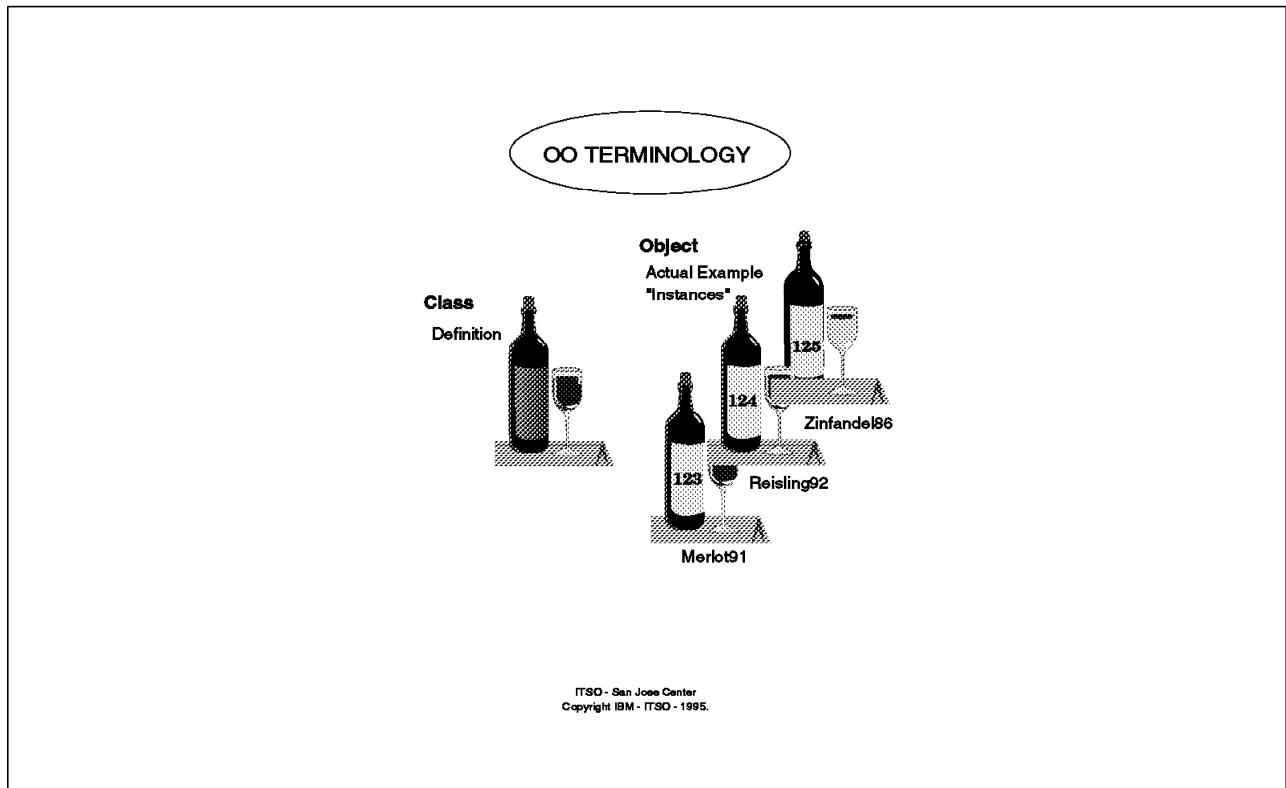


Figure 12. Class-Object Relationship

One object communicates with another via a message as shown in Figure 13. For example, the wine application has objects "wine-type" and "grape-type". To check if we can drink a certain bottle with a piece of food, we send a message to our "wine-bottle" object specifying the method "wine-bottle OK with food ?" together with a piece of data "roast beef". The wine-bottle object's method then sends a message to its "wine-type" object saying, say "wine OK with food ?" and the same data "roast beef". The "wine-type" object then sends a message to the "grape-type" object saying "grape-type OK with food ?" and "roast beef". Finally, the "grape-type" object checks its list of acceptable foods and sends a message back, this time just with some data.

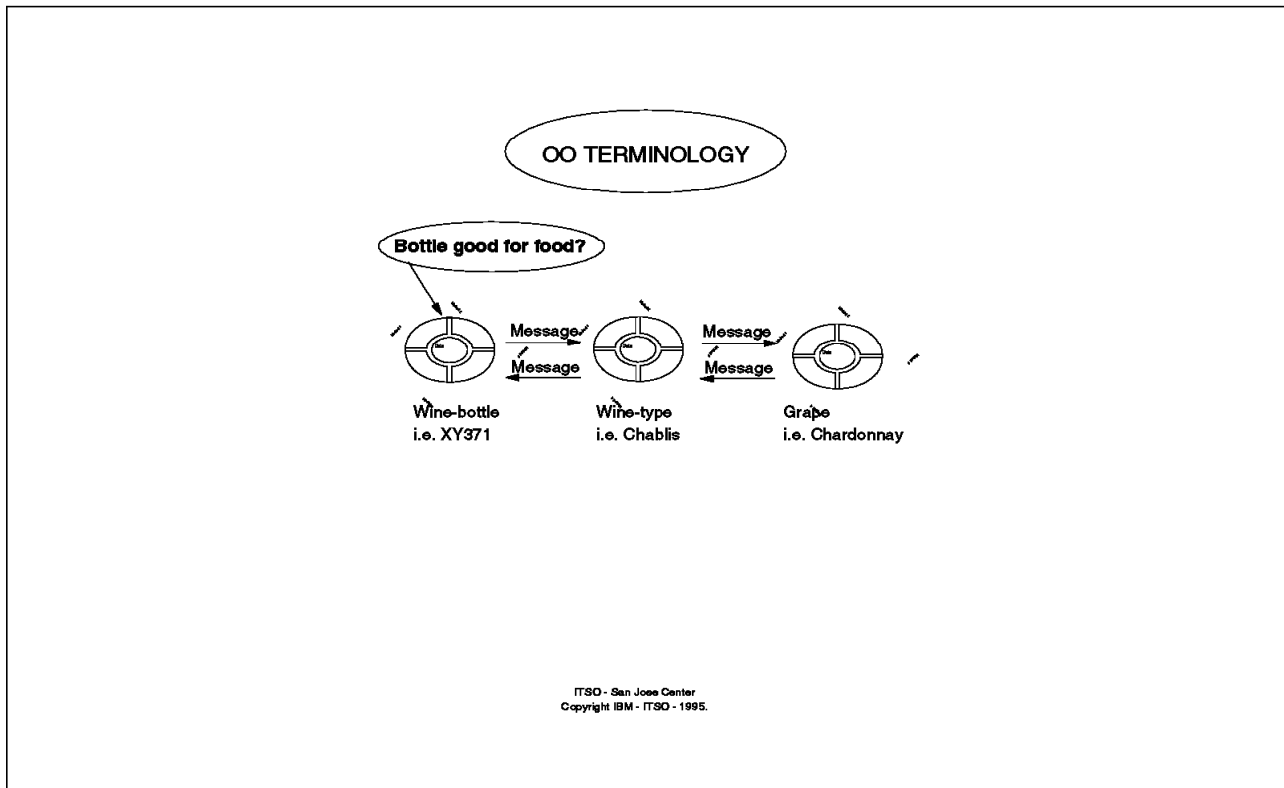


Figure 13. Messages

This example is useful for distinguishing classes from objects. There is just the one class "grape-type" but there are many "grape-type" objects: one for Cabernet Sauvignon, one for Merlot, one for Zinfandel, one for Chardonnay, and so on.

In this example "wine-bottle" object represents a bottle of Chablis, made from the Chardonnay grape. Accordingly the "RP987" object (of the "wine-bottle" class) sends a message to the "Chablis" object (of the "wine-type" class) which sends a message to the "Chardonnay" object (of the "grape-type" class).

So, the basic terms used to discuss object-oriented are summarized in Figure 14. It really is as easy as PIE!

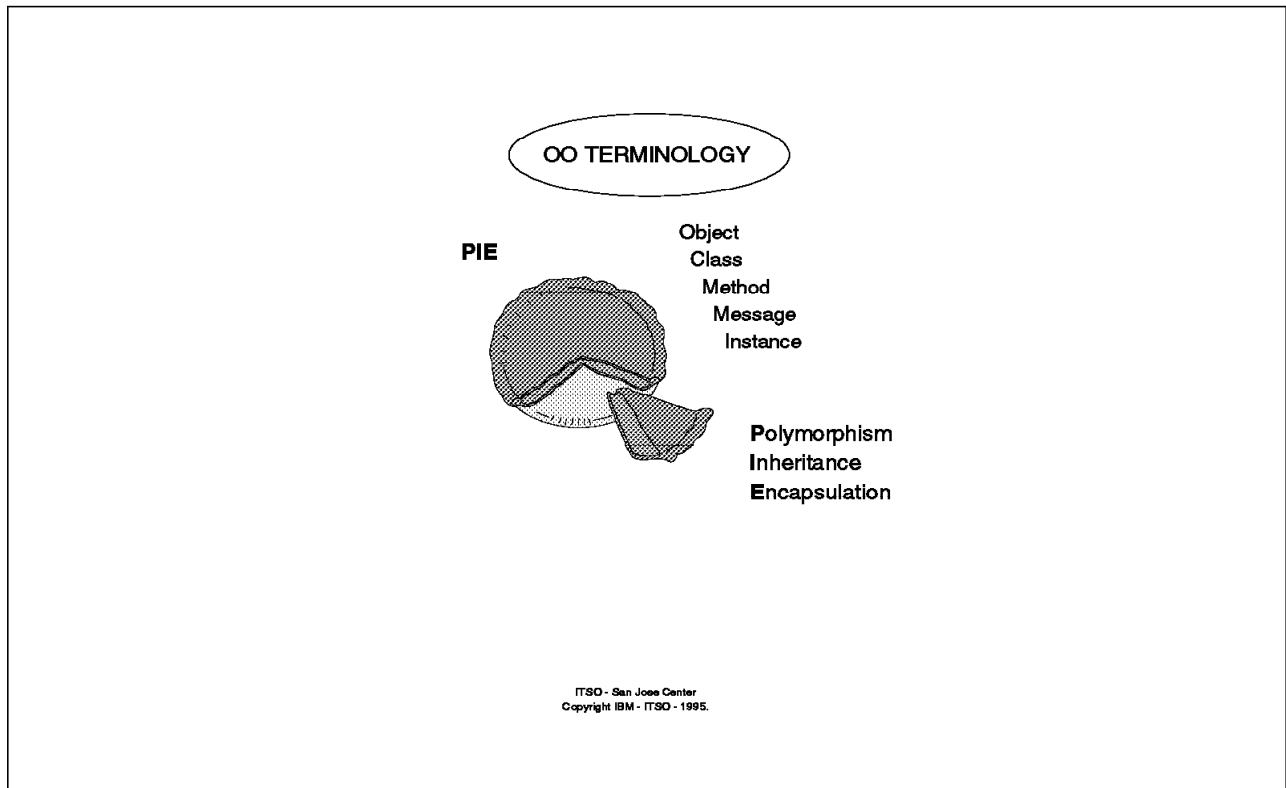


Figure 14. OO Terminology

## 2.6 OO Review

Figure 15 reviews the claims of object-oriented programming and how the various components of object-oriented contribute.

Re-use comes from the basic object identity and the ability to obtain slight modifications *without copying* by "inheritance".

User communications come from the "real-world" nature of objects compared with the artificial abstracted model traditionally used.

"Encapsulation" leads to data integrity.

"Polymorphism" leads to direct productivity and elapsed time reductions.

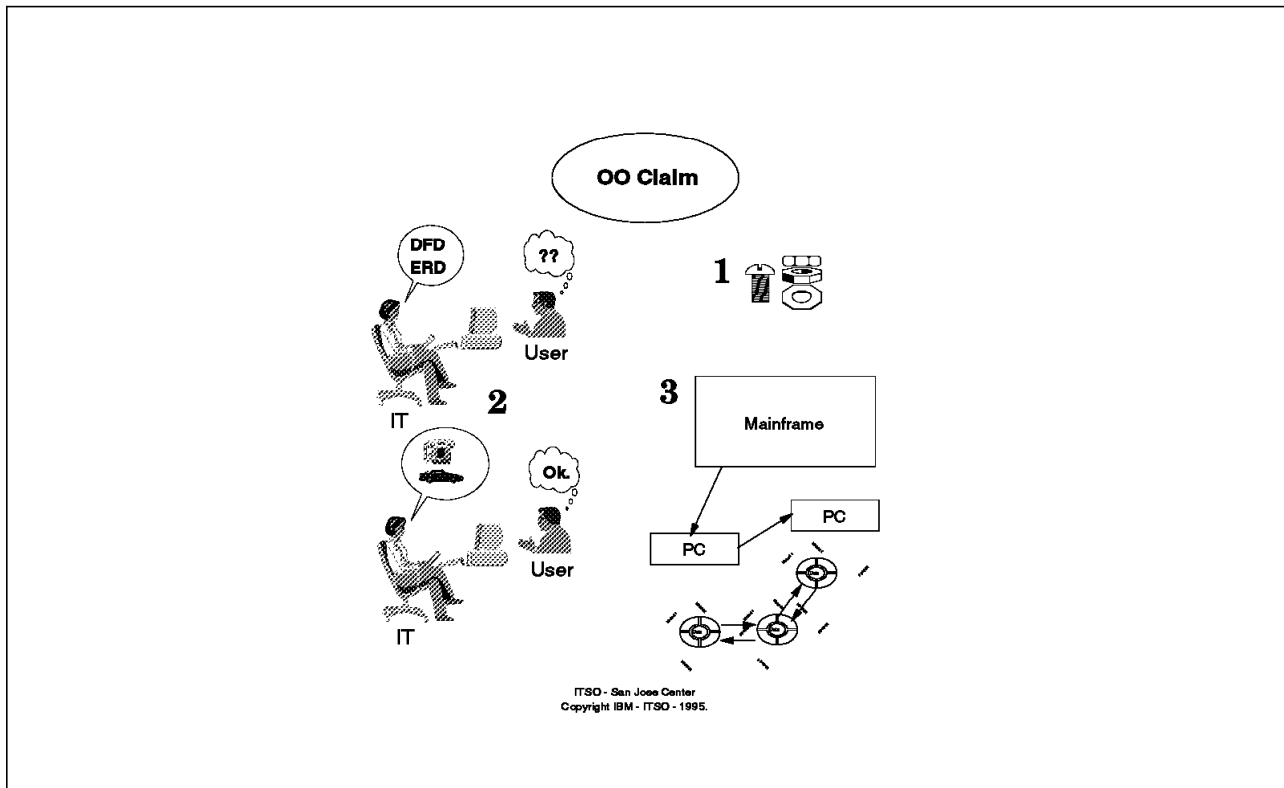


Figure 15. How OO Scores

## 2.7 OO History

OO was first discussed by a group of people working on the SIMULA language in the late 1960s. Xerox PARC researchers developed Smalltalk-80 in the late 1970s. Bjarne Stroustrup developed C++ for Bell Labs in 1981.

Taligent was founded as a joint venture between IBM and Apple (later an investment made by Hewlett-Packard). This development is illustrated in Figure 16.



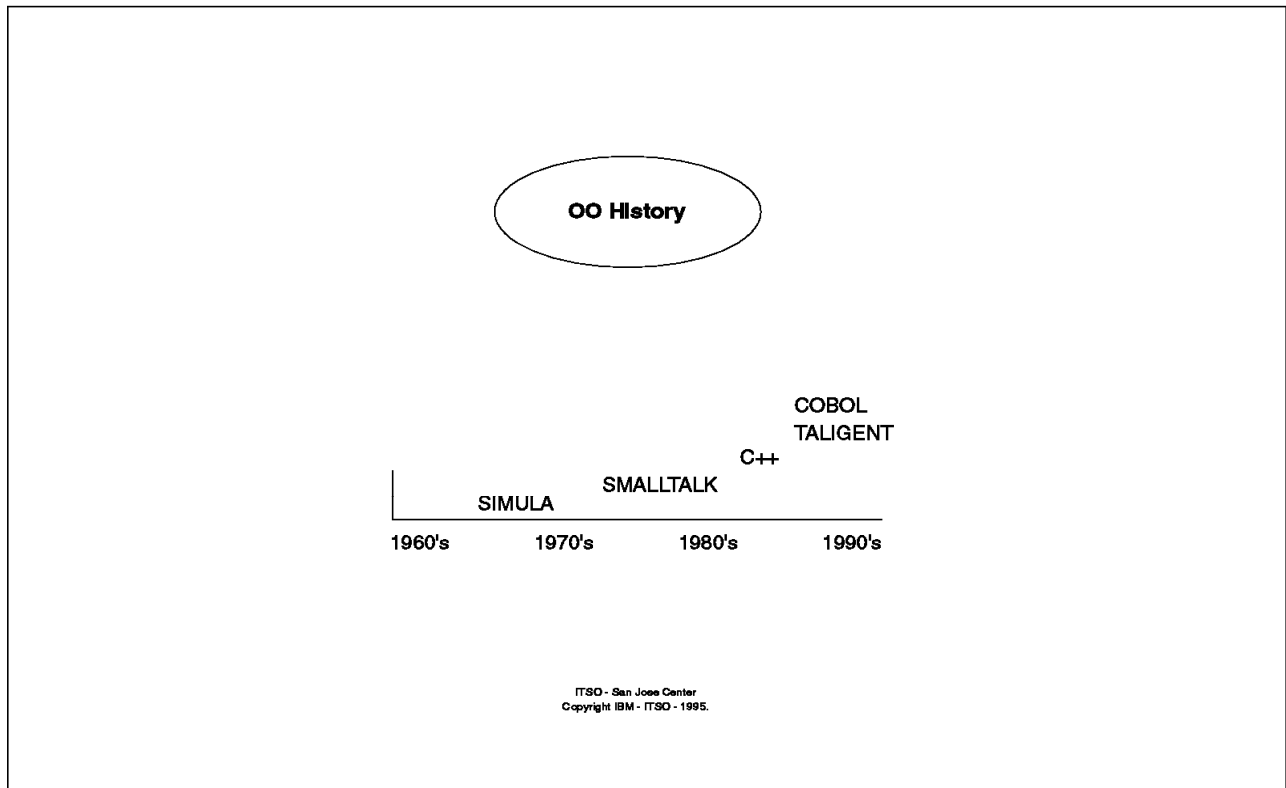


Figure 16. OO History

In 1989 a meeting in Scottsdale, Arizona involving a number of COBOL and object-oriented interested parties recommended further work on combining the two technologies. As a result, a task force was set up, now called the X3J4.1 group, who in 1993 presented their first proposals to ANSI.

The reasons for the strange name is that the COBOL standards group is known as X3J4. The corresponding ISO group, the international COBOL standards committee, is called ISO/IED JTC1 SC22/WG4, or WG4. WG4 has subcontracted the work of developing the new COBOL standard to X3J4. The relationships are shown in Figure 17.

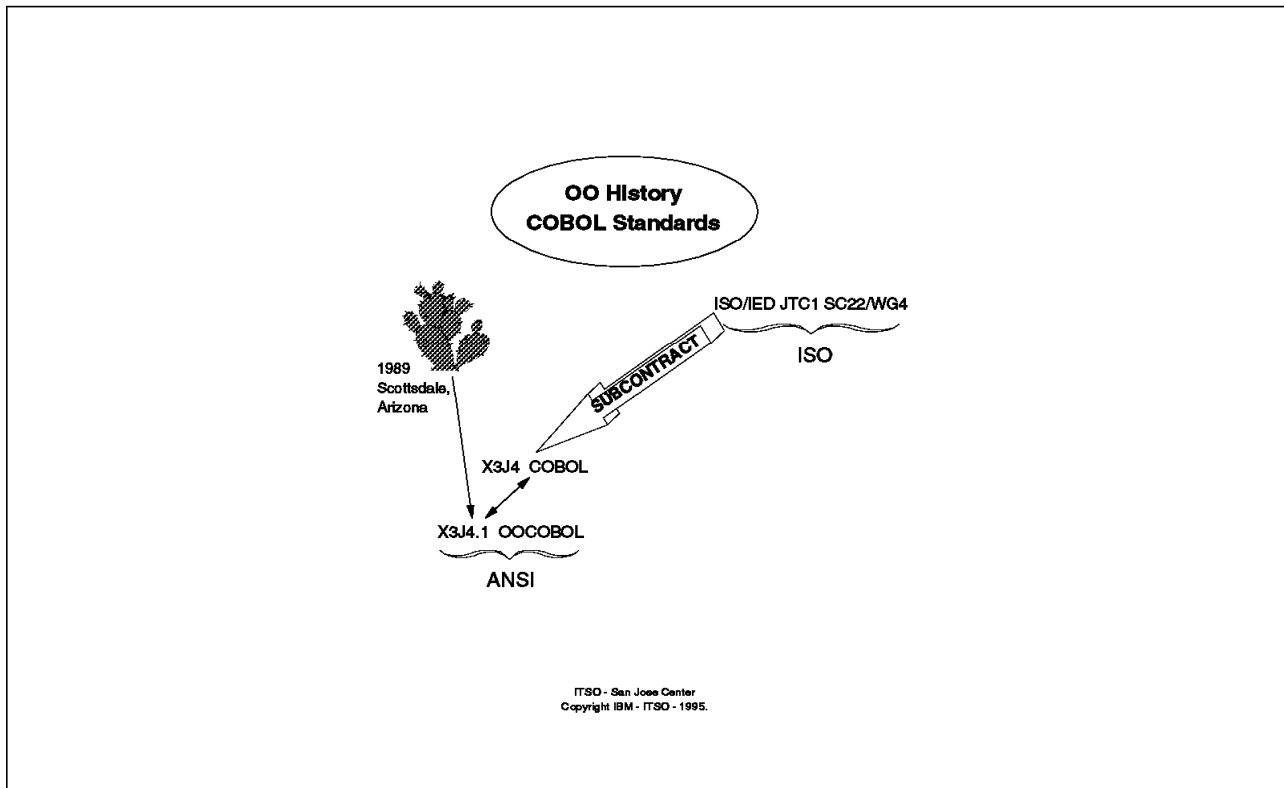


Figure 17. OO COBOL Standards

One of the attractions of object-oriented programming is the prospect of being able to buy many classes off-the-shelf, assembling them together to make a unique system from a collection of standardized parts. To achieve that goal, the key word is "standardized". For that reason in the late 1980s a number of software manufacturers formed an organization devoted to establishing the standards required. The group is called the Object Management Group (OMG) as referred to in Figure 18.

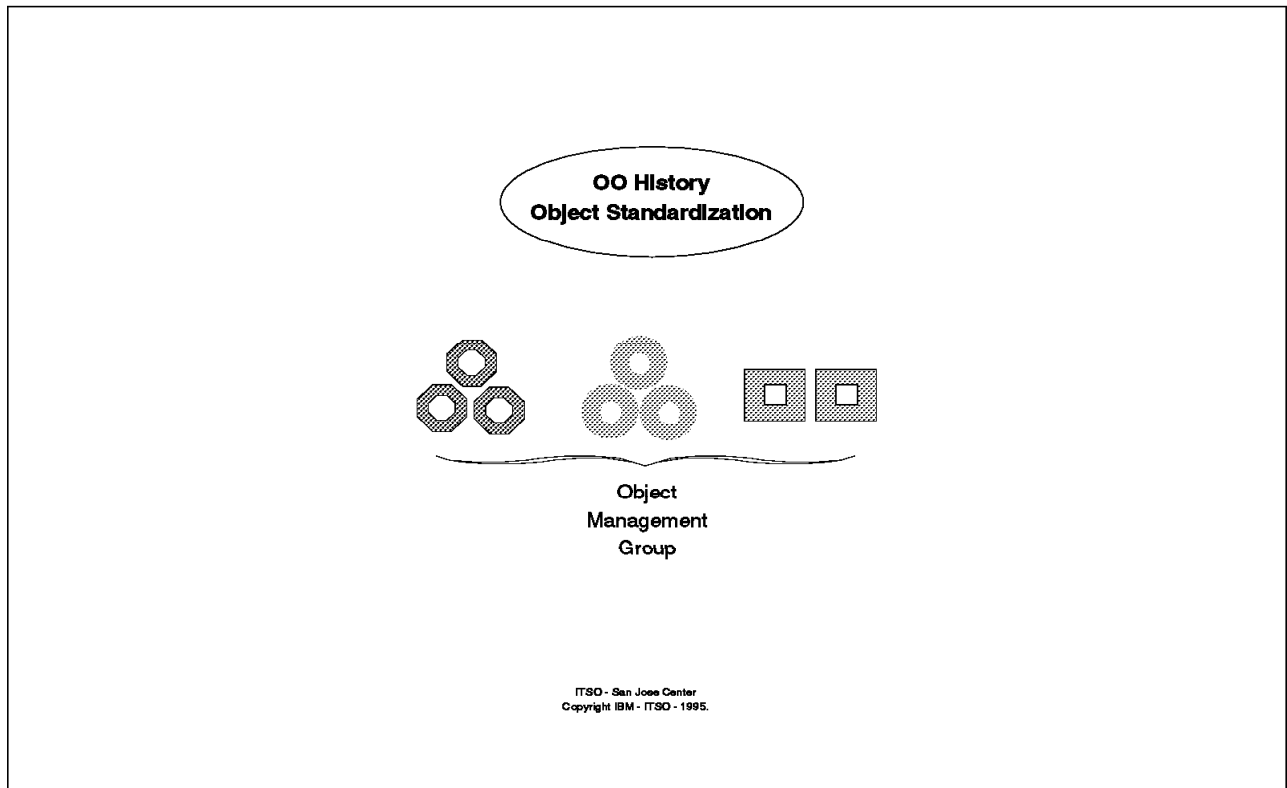


Figure 18. Object Standardization



## Chapter 3. Old COBOL Approach

This chapter explains the traditional COBOL approach to programming applications.

### 3.1 OO Image

Many people, even those involved with OO, think of it as a workstation-based technology. The image refers not to the actual platform but to the small scale nature of its operation.

However this is not so. One of the best OO references is the Brooklyn Union Gas Company that rewrote their Customer Information System (CIS) between 1987 and 1990 in an object-oriented way as referred to in Figure 19. This CIS manages a \$1 billion revenue stream, uses a 100 gigabyte database, has over 400 screens with 850 users, and processes 350,000 transactions per day at an average rate of ten transactions per second.

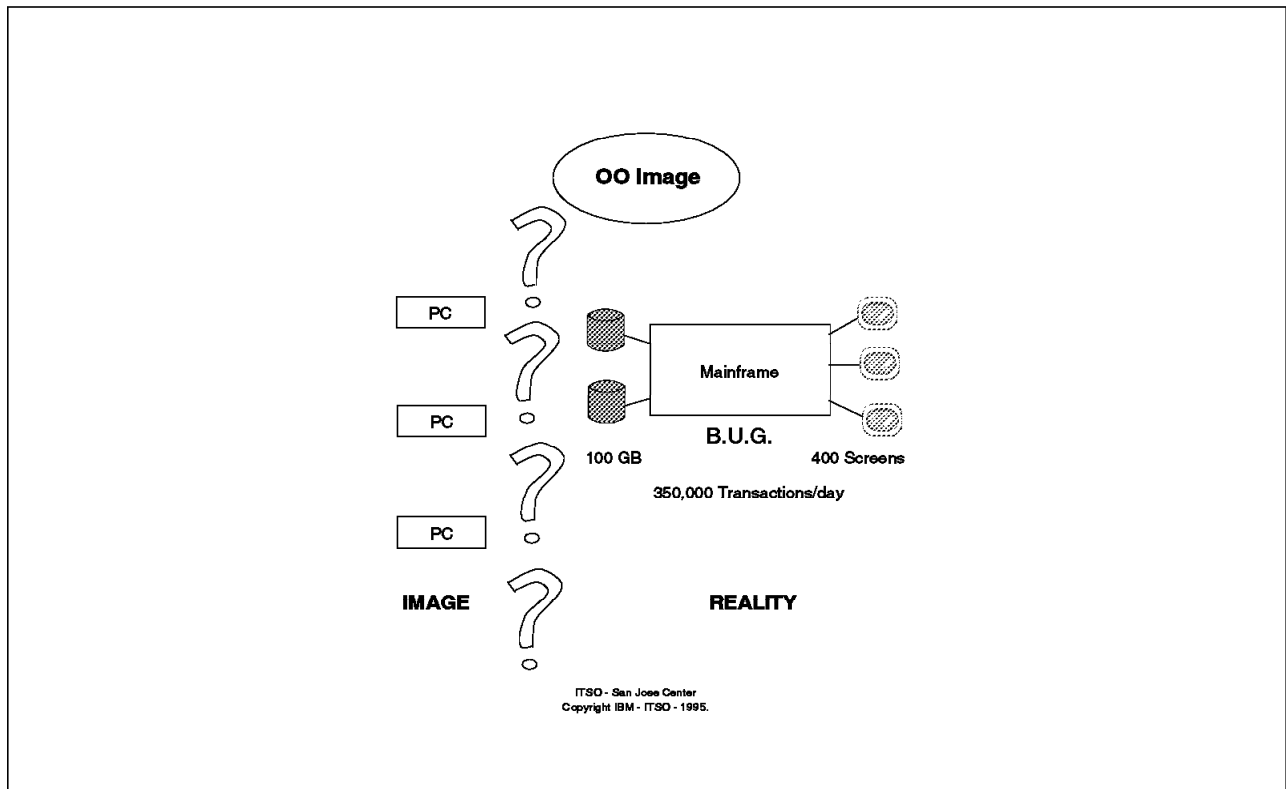


Figure 19. OO Image versus Reality

This is a successful mainframe object-oriented application, using PL/I. Could it have been done using COBOL?

## 3.2 Encapsulation

With COBOL the concept of "encapsulation" can be used by structuring a program as illustrated in Figure 20. The data is defined in the program and is invoked, or "called." The program passes one parameter indicating what it wants, and the second parameter contains the relevant data.

Thus the program is the object, and the data is accessed only by routines in this program.

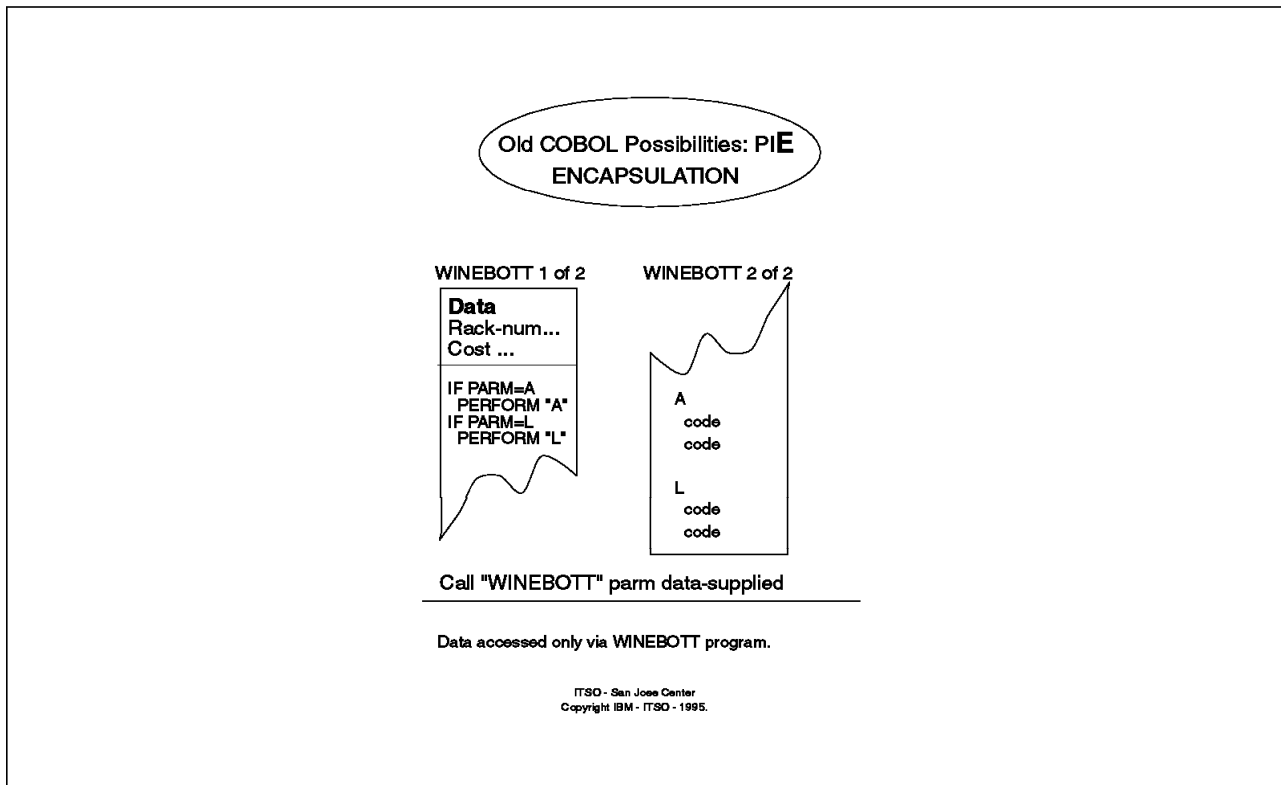


Figure 20. Encapsulation

COBOL can provide inheritance too as shown in Figure 21. We define a second program which contains the extra routines not in the original program. Our second program tests to see if it is being asked to perform one of its extra methods. If so, it does. If not it calls the original program passing the parameters it itself received.

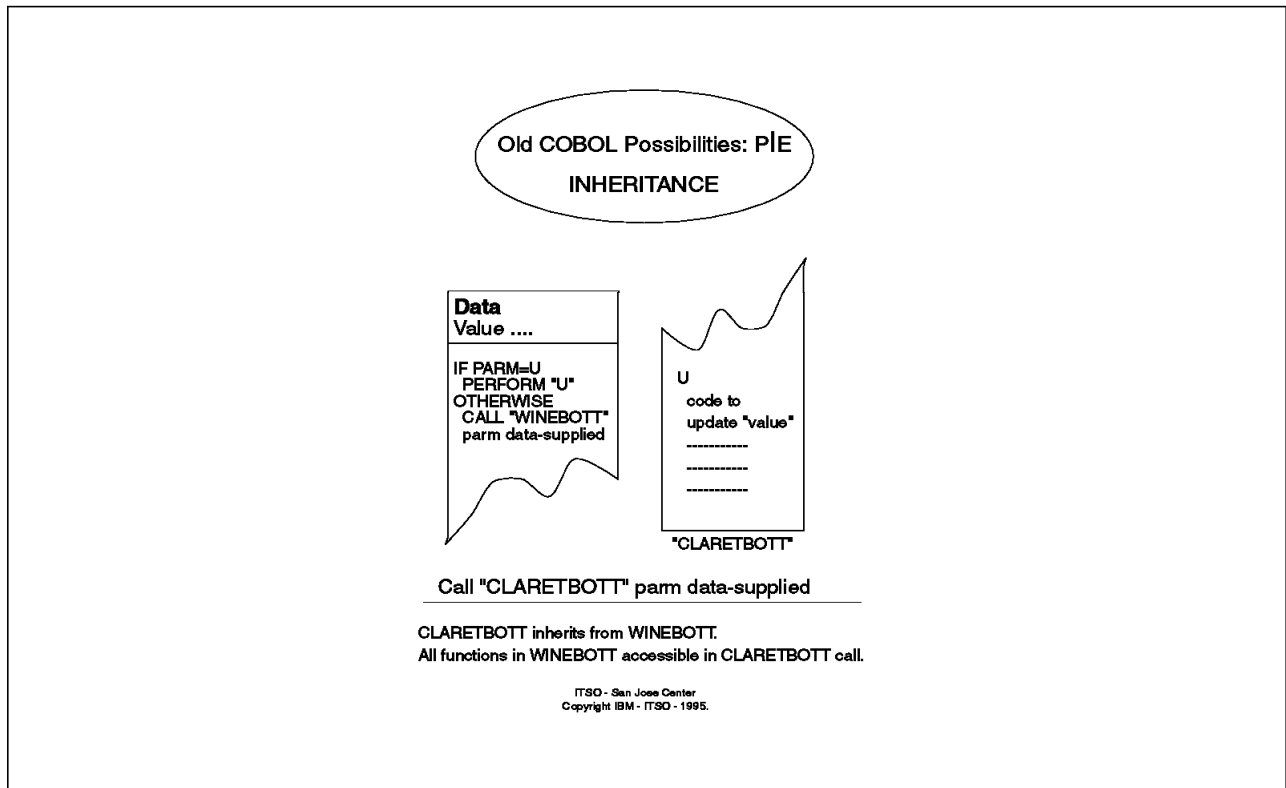


Figure 21. Inheritance

This second program needs to contain nothing about how the original program works, or even what it can and can't do. If an invalid request has been made, the first program handles it just as if it had been sent the request directly, that is, it is transparent to the second program.

If we want to change a standard method or to add a new one, we can do that by changing the WINE-BOTT program. CLAR-BOTT picks the change up without having to change or recompile it at all.

Polymorphism is provided by COBOL with a similar mechanism as shown in Figure 22. In our second program we define a new version of the routine already defined in the original program. If the second program (that is, the second, or inheriting object) is invoked with this method, it is the second program's routine that gains control.

When the first program is invoked, its routine is the one that runs.

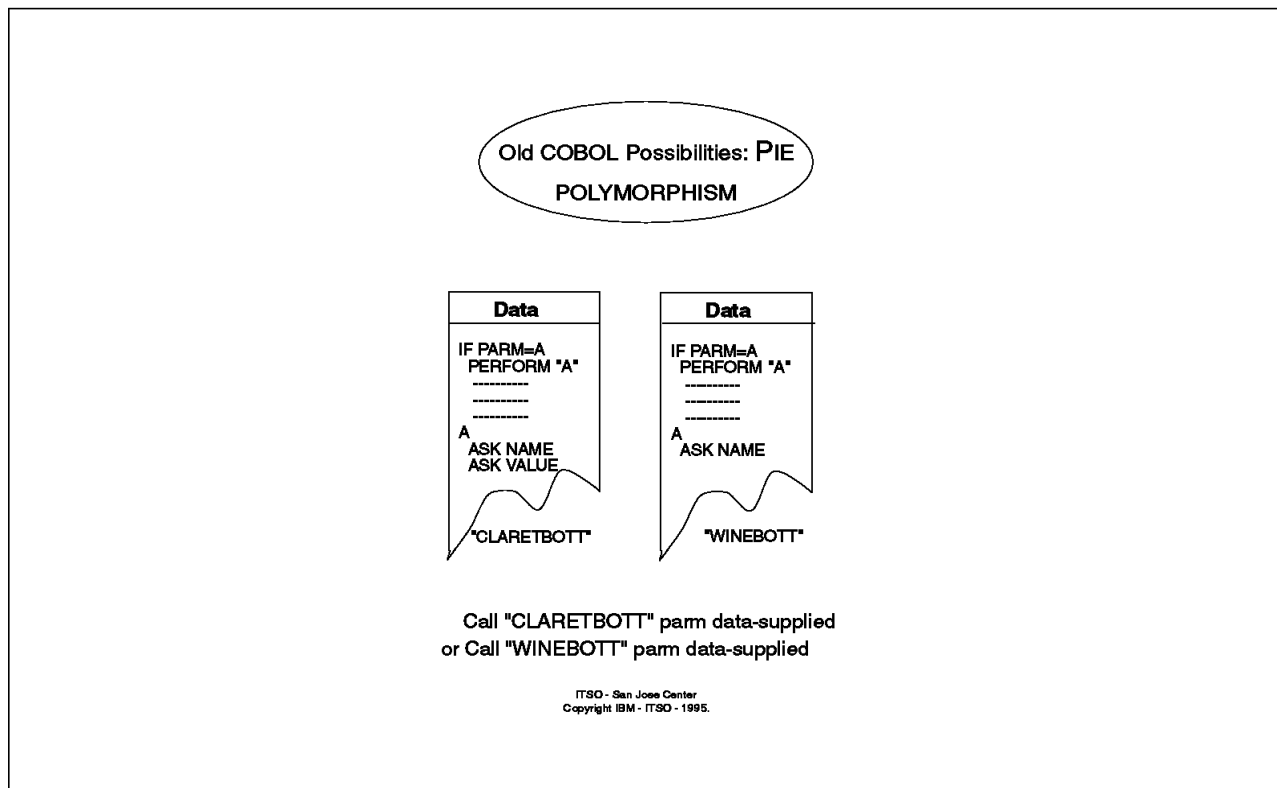


Figure 22. Polymorphism

### 3.3 Approach Limitations

There are, however, a number of problems with this approach as summarized in Figure 23. Some of these problems can be described succinctly only by reference to COBOL syntax, however, here are some high-level observations.







---

## Chapter 4. New COBOL Approach

This chapter explains the new COBOL approach to object-oriented programming.

---

### 4.1 Requirements

Figure 24 illustrates the requirements for the new COBOL approach. We need to add facilities to either the language or the environment to address the points in Figure 24. These can be summarized as allowing explicit definitions, messaging, keeping the one copy of object data, and adopting a defined, standardized approach.

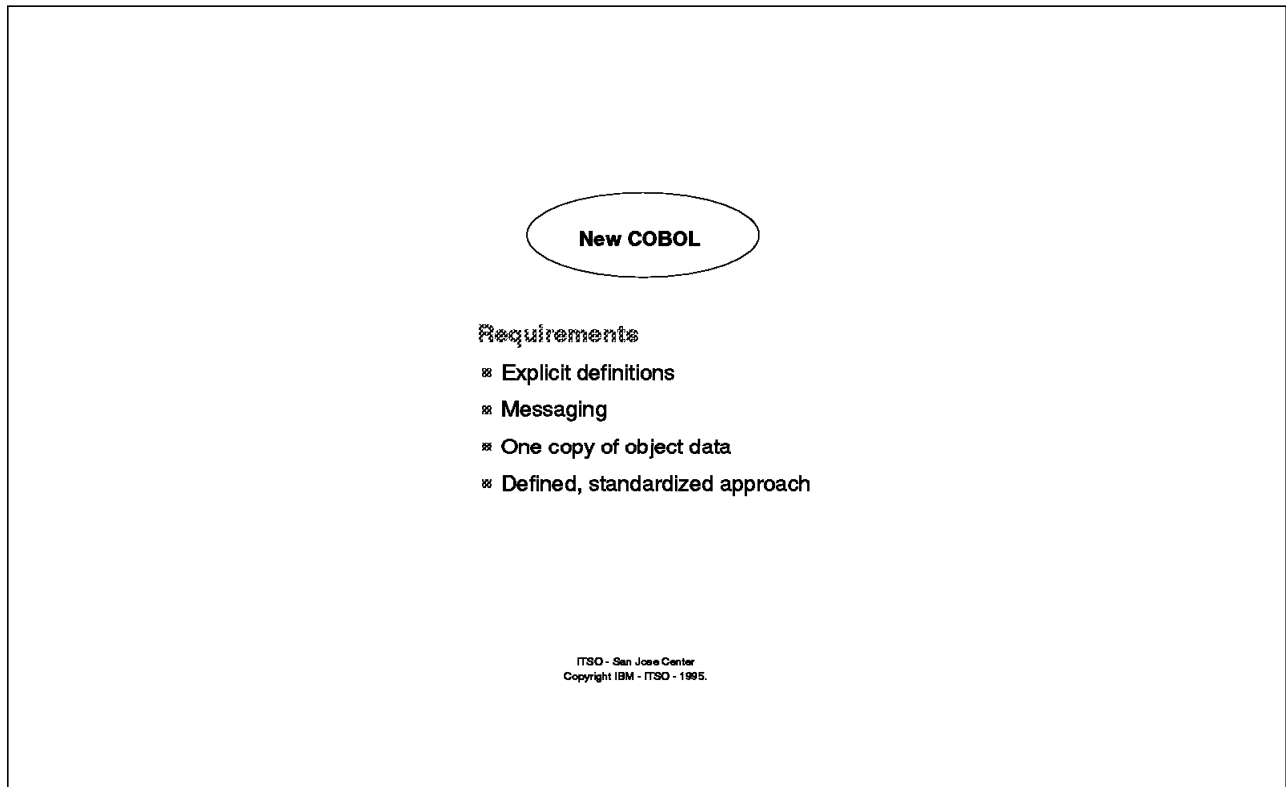


Figure 24. OO Requirements

---

### 4.2 Explicit Definition

In OO COBOL, we define classes with programs. Each program defines one class, its data, and its methods. Each method is a mini-program, containing its own data if needed, as shown in Figure 25.

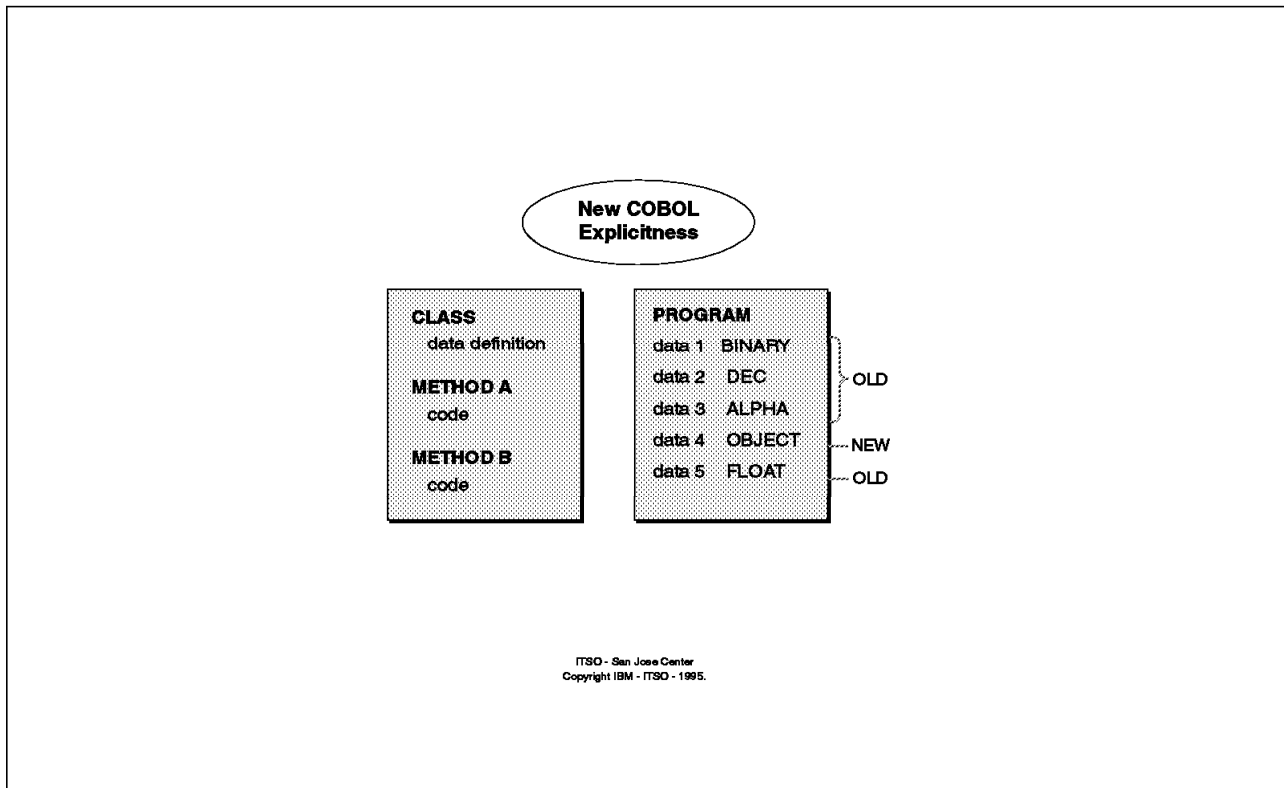


Figure 25. Explicitness

Just as in old COBOL we can refer to data of different "types," such as decimal, binary, alphanumeric, and so on, in OO COBOL we have a new data type, the Object Reference. This means that we can keep it in one place, and point to it from everywhere else. But we still can access its data only with its methods. This has the additional benefit that we can work on the object as an object, for example to compare it with another.

### 4.3 Client/Class Program Relationship

A simple object system is shown in Figure 26. There is one "client" program which forms a liaison with a number of objects to achieve its required results. The objects themselves do this with other objects.

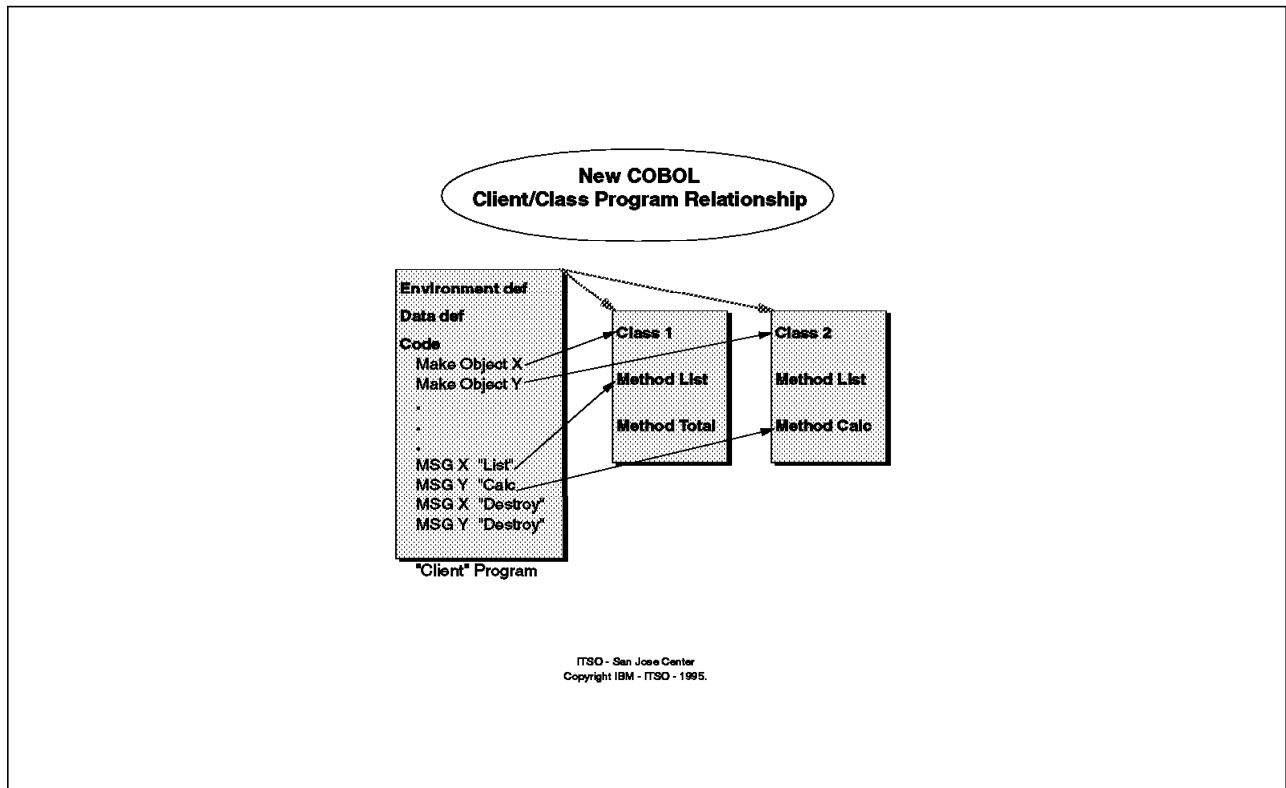


Figure 26. Client/Class Relationship

The sequence is as follows:

1. The client program refers to the classes whose objects it wants to contact, thus connecting the logic of the program to the reality of the environment. This is similar to identifying the files to be used.
2. For every object it wants to use it sends a message to the object's class saying the equivalent of "make me an object".
3. It sends messages to the objects as its logic requires, indicating things such as "initialize yourself," "tell me your attribute NAME".
4. On receiving these messages the objects can send their own messages to other objects.
5. Once the "client" program finishes, it sends a "destroy yourself" message to the object.

In the sequence above we have used a "client" program. But this program itself could be an object, initiated by the system itself on start-up.

## 4.4 Recursion

One powerful and required new facility is the ability of the objects to send messages to themselves or even allowing a method to call itself. This technique is called recursion as shown in Figure 27. The traditional example quoted in computing is the calculation of factorials.

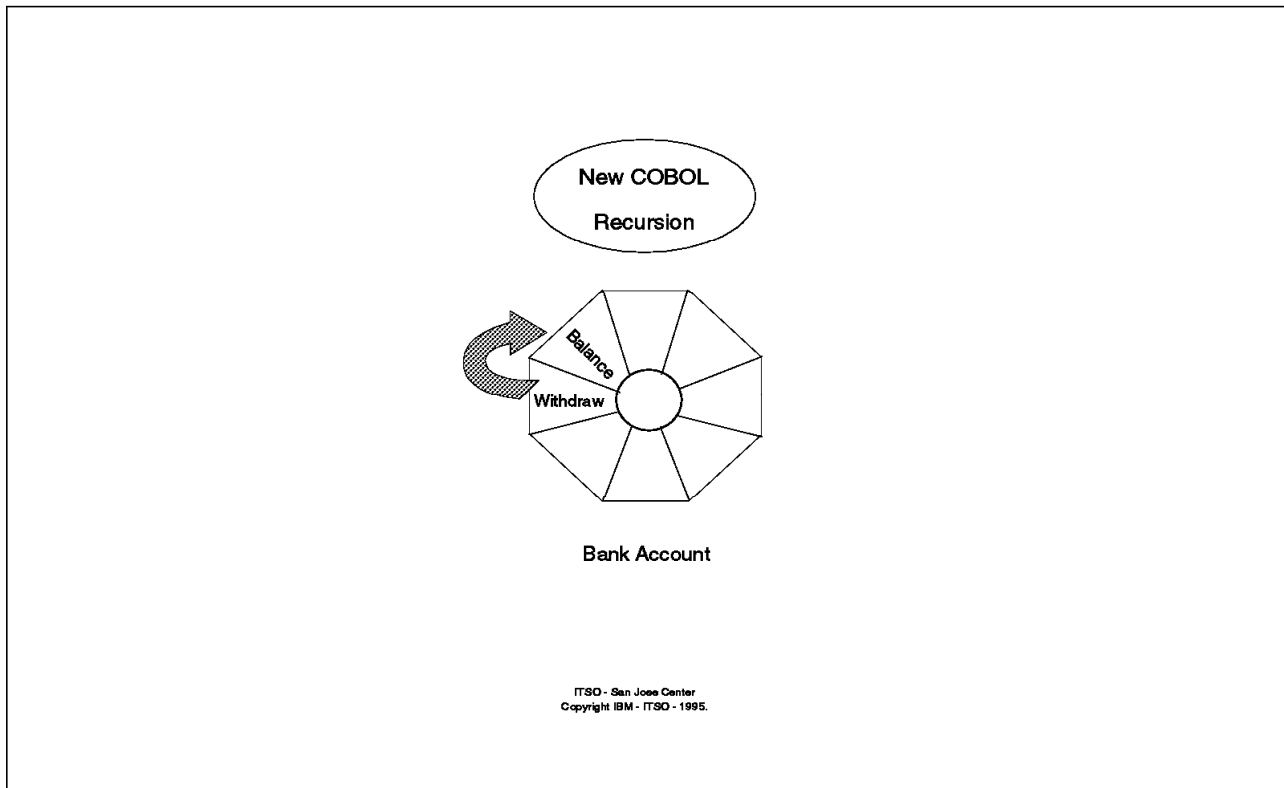


Figure 27. Recursion

Another example is the bank account object sending a "Withdraw \$100" message and sending itself a message "Give balance" first to check that there is indeed enough money in the account to allow the \$100 to be withdrawn.

## 4.5 Summary of Language Changes

The additional functionality of the new COBOL approach requires minimal changes. This is welcome news to those who have migrated from OS/VS COBOL to a later version. The required new statements are shown in Figure 28.

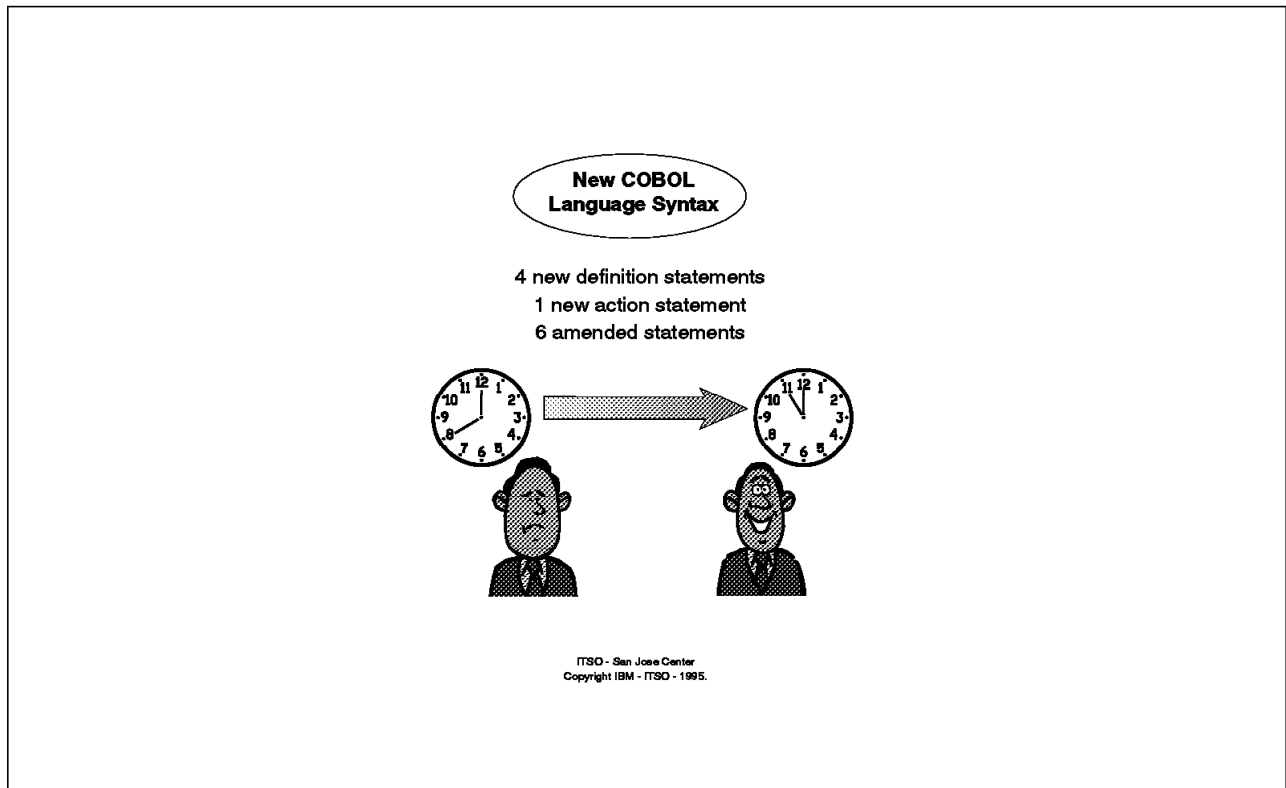


Figure 28. Syntax

A new statement allows the definition of a class, another for a method, another for the method-specific data, and several new statements allow reference to other classes. A new statement allows programs to send messages to objects (the "invoke" verb).

Several existing statements have only simple additions. A COBOL programmer should have no difficulty in learning all the new syntax. Learning to program in OO requires much more than familiarity with the correct syntax.

## 4.6 ANSI Standards

There have been three major ANSI standards and a significant "addendum" in 1989 which added "intrinsic function" support. The next standard is expected to be agreed in 1997 and a major part (though only a part) are definitions of how OO should be supported. This is illustrated in Figure 29. Since the standard has not yet been established, how is it possible to bring out a product with OO constructs?

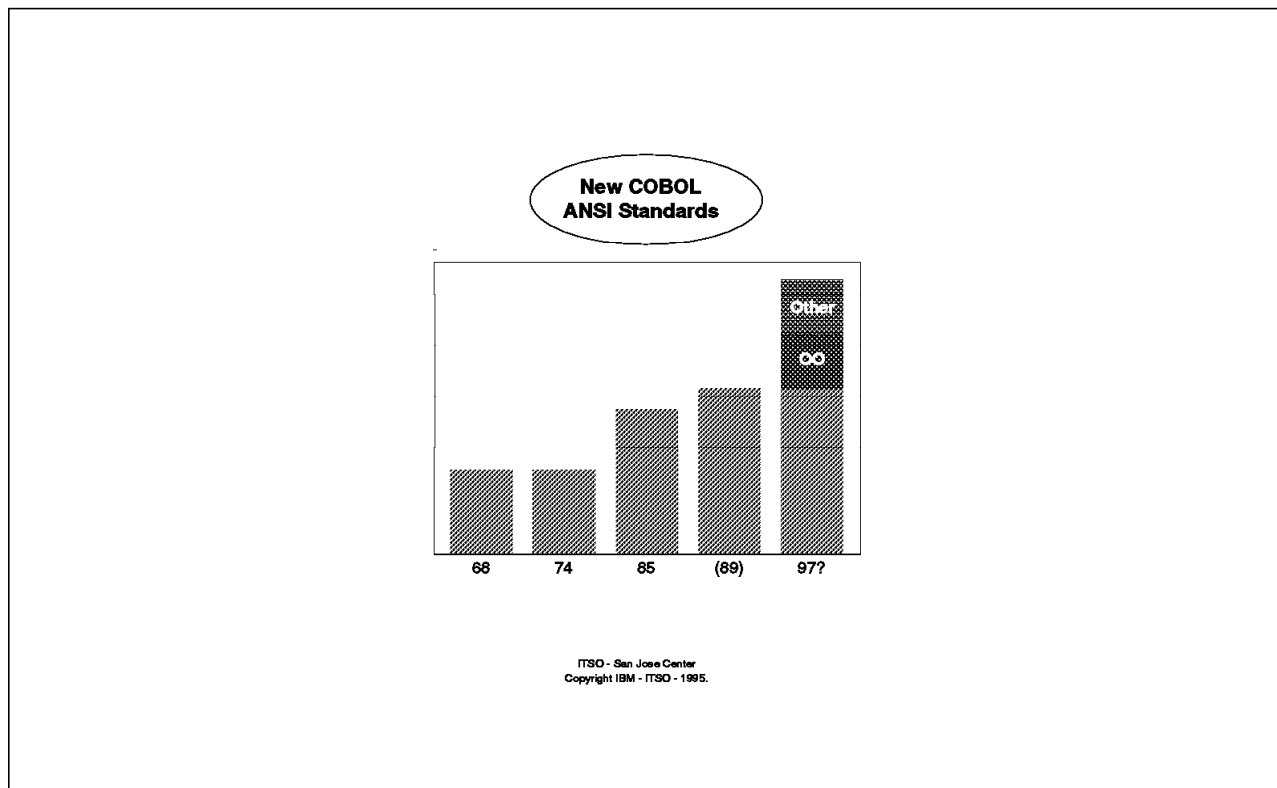


Figure 29. ANSI Standards

There is a sufficient central core of standards agreed upon between the committee members (IBM being a major player) to enable a product to be produced with support for that subset. In other words, the OO functions (as described above) are only a subset of what will eventually be in the standard.

As mentioned earlier, object-oriented programming has existed for some time but with many inhibitors. Several of them, shown in Figure 30, are addressed by IBM's System Object Model (SOM) which forms one of IBM's most significant differentiators from our object-oriented competition.



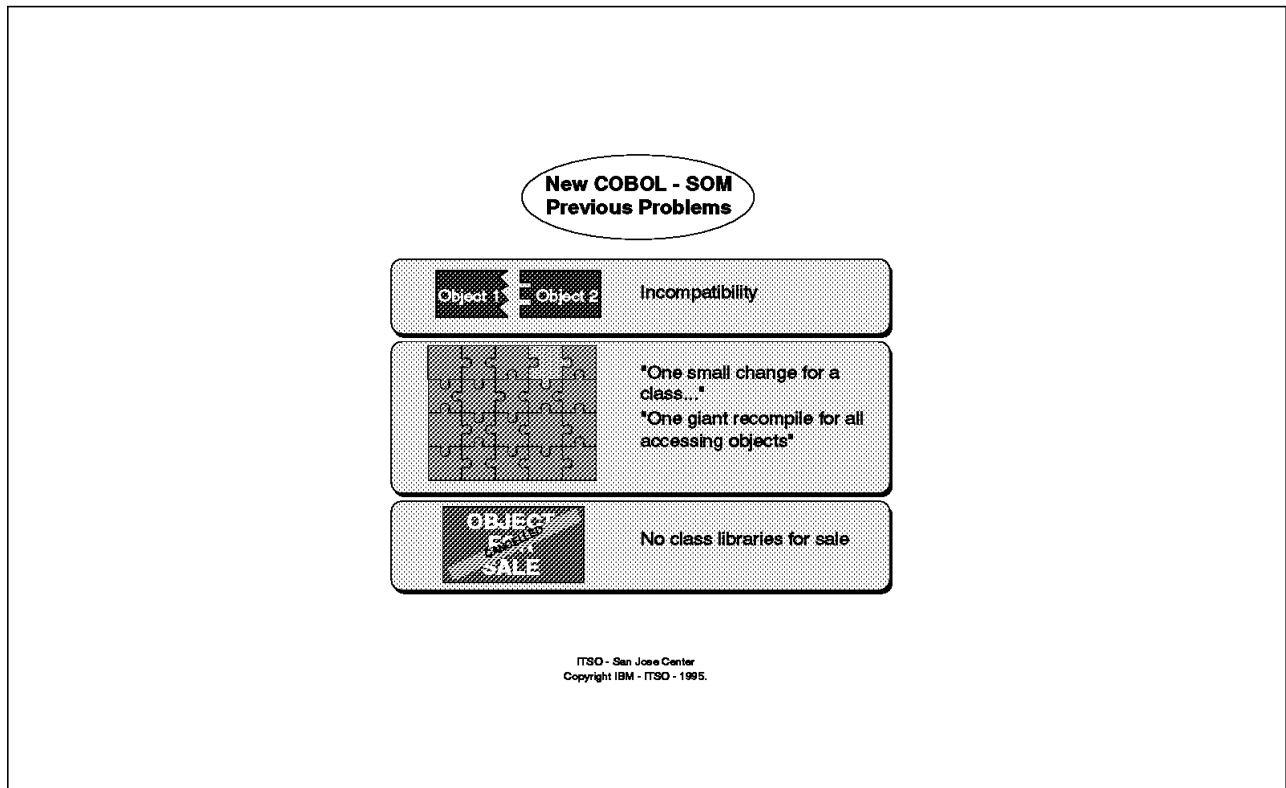


Figure 30. Problems without SOM

One problem has been incompatibility, not only between one language and another but between the same language provided by different suppliers. In particular, an object created using C++ from supplier B is not necessarily compatible with a C++ object from supplier M.

Another problem is maintenance of classes. If a program accessed an object from a class, and then that class has an extra method added, the original program often needs to be recompiled, even though the change made had no connection at all to the original relationship.

SOM provides:

- A mechanism for defining classes and class libraries
- Object management services.

In other words, SOM is an infrastructure which handles all aspects of object-oriented systems support, including the creation and deletion of objects, the routing of messages from one object to another, and all other runtime requirements as illustrated in Figure 31. The actual definition of the classes and the writing of the programs to use them is the programmer's responsibility.

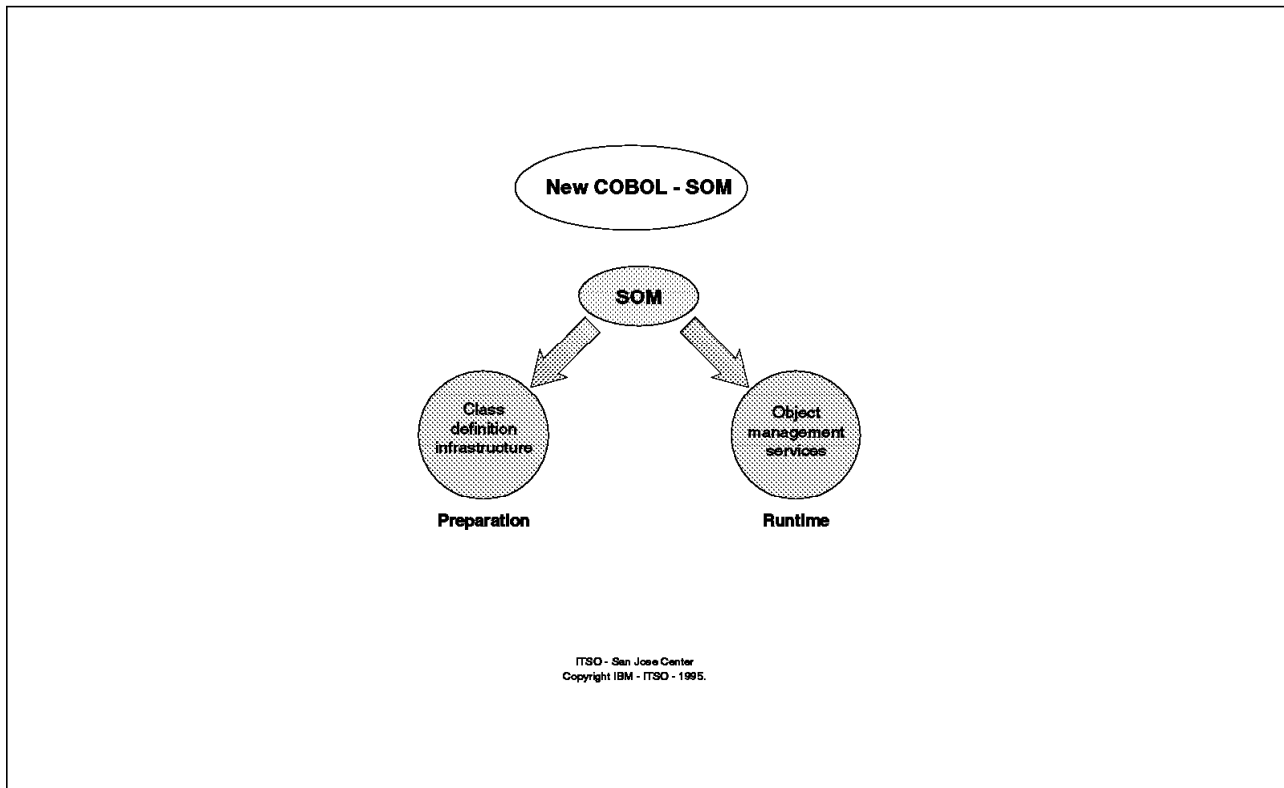


Figure 31. SOM Definition

A rough analogy might be CICS. The programmer is still responsible for writing the program but after that CICS handles the details, such as the registration of the program, its invocation when required, its memory management. In addition CICS does not care whether the program is PL/I, COBOL, or C.

With SOM, it is possible to use compilers for different languages from different suppliers. It is not necessary to recompile programs when access methods change (assuming that the change does not directly affect that program). Software companies can now offer Class Libraries, secure in the knowledge that their code cannot be copied, but still allow the user ability to use and inherit from what they have provided.

The key to the SOM breakthrough is the separation of implementation details from interface definition, as shown in Figure 32.

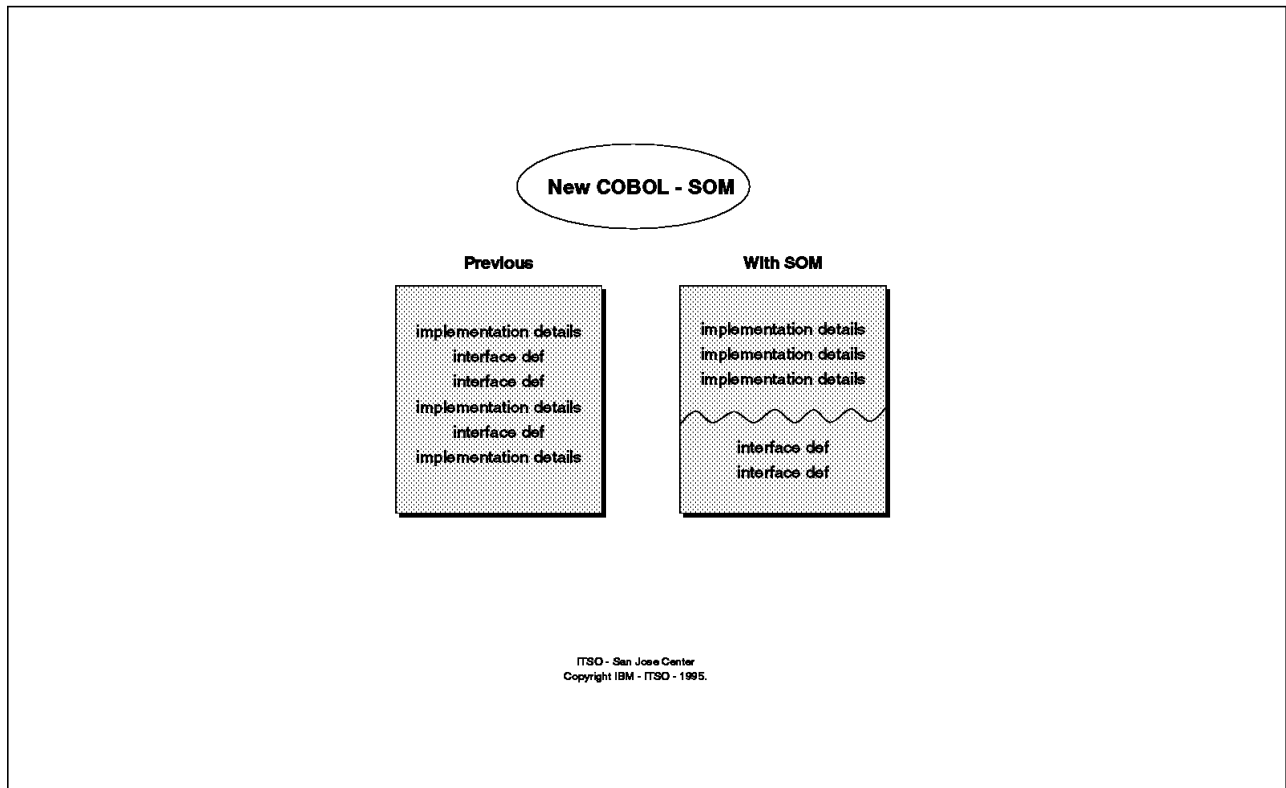


Figure 32. How SOM Works

Implementation means what the methods actually do. Interface means the names of the methods, how many parameters they take, and what data types they are, not what they do.

Logically, all we need to do to interact with an object is to have the interface details. Everything else is a black box. SOM provides that black box.

**Platform availability:** SOM was originally available on the OS/2 platform only. But it is now available on AIX, Microsoft Windows, and MVS as shown in Figure 33. Classes defined to SOM on one platform can be readily ported to any of the other platforms.

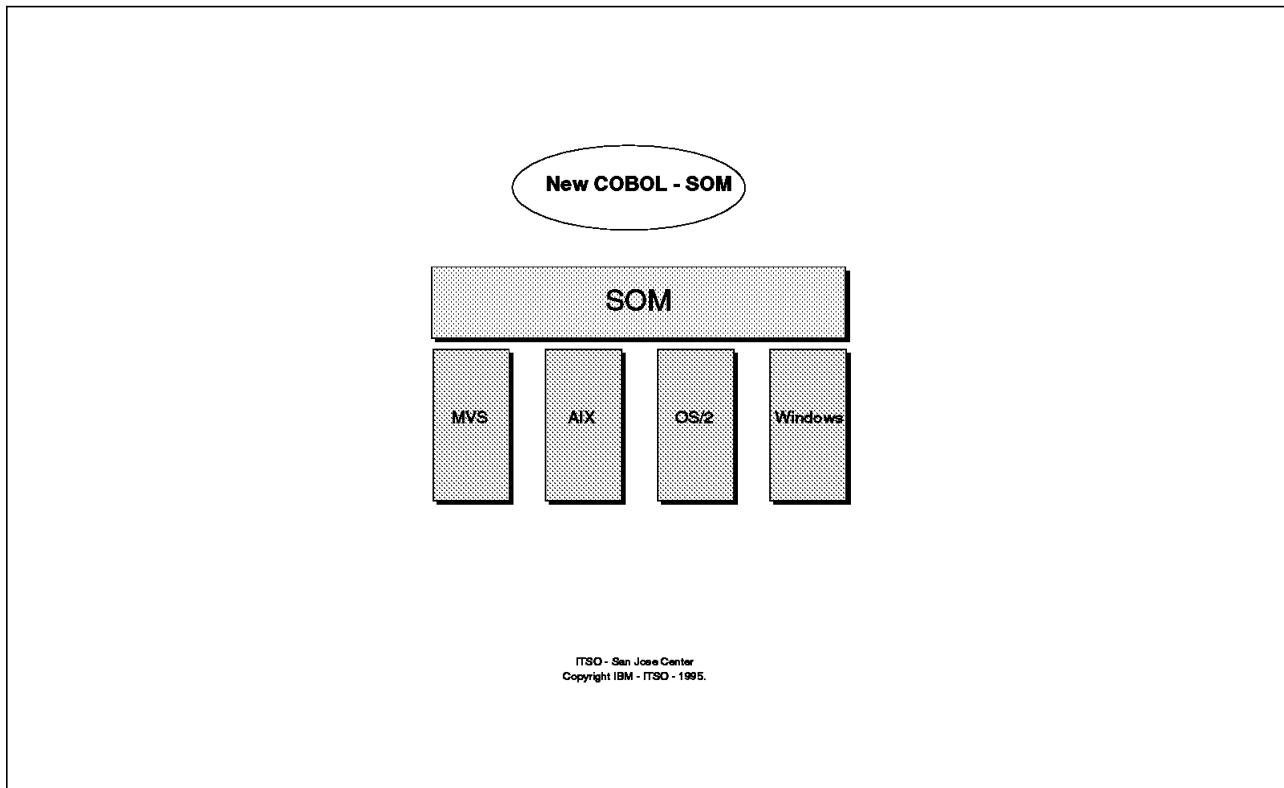


Figure 33. Platform Support

**DSOM:** In 1991 the Object Management Group (OMG) published a specification of the architecture to be used in developing the mechanism which objects use to communicate with each other. This is called Common Object Request Broker Architecture (CORBA). SOM is IBM's implementation of that architecture *on one platform* as shown in Figure 34.

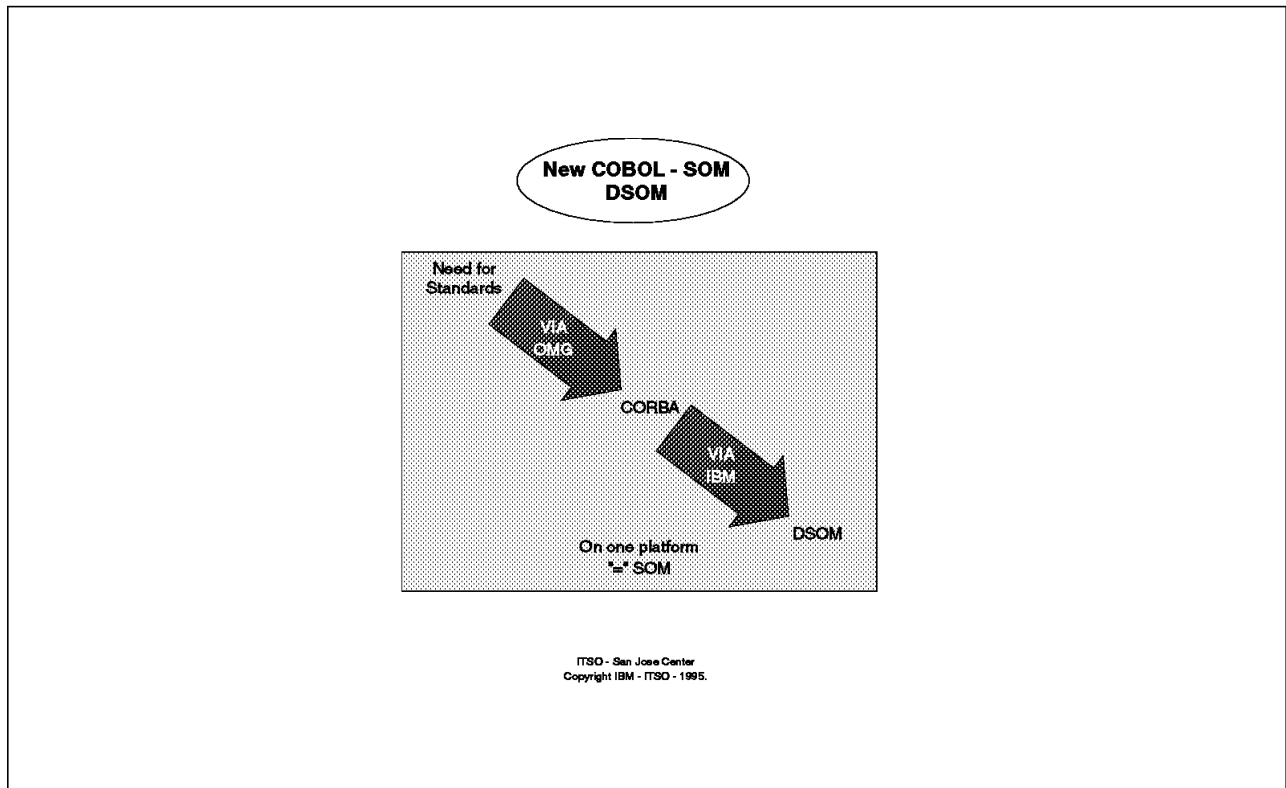


Figure 34. Som and DSOM

CORBA allows objects to communicate across different platforms (as well as on the same platform). In IBM's case this compatibility is provided by Distributed SOM (DSOM). The methods that DSOM offers are the same as SOM's. Any SOM object can be moved to another platform without changing its code.

DSOM is already available on OS/2, AIX, and Windows and is a statement of direction for MVS.

There is very little of SOM of which the COBOL programmer has to be aware as shown in Figure 35. The COBOL compiler interacts with SOM in an automatic manner so that as a by-product of compilation, all SOM's required information is stored in SOM's Interface Repository, its database.

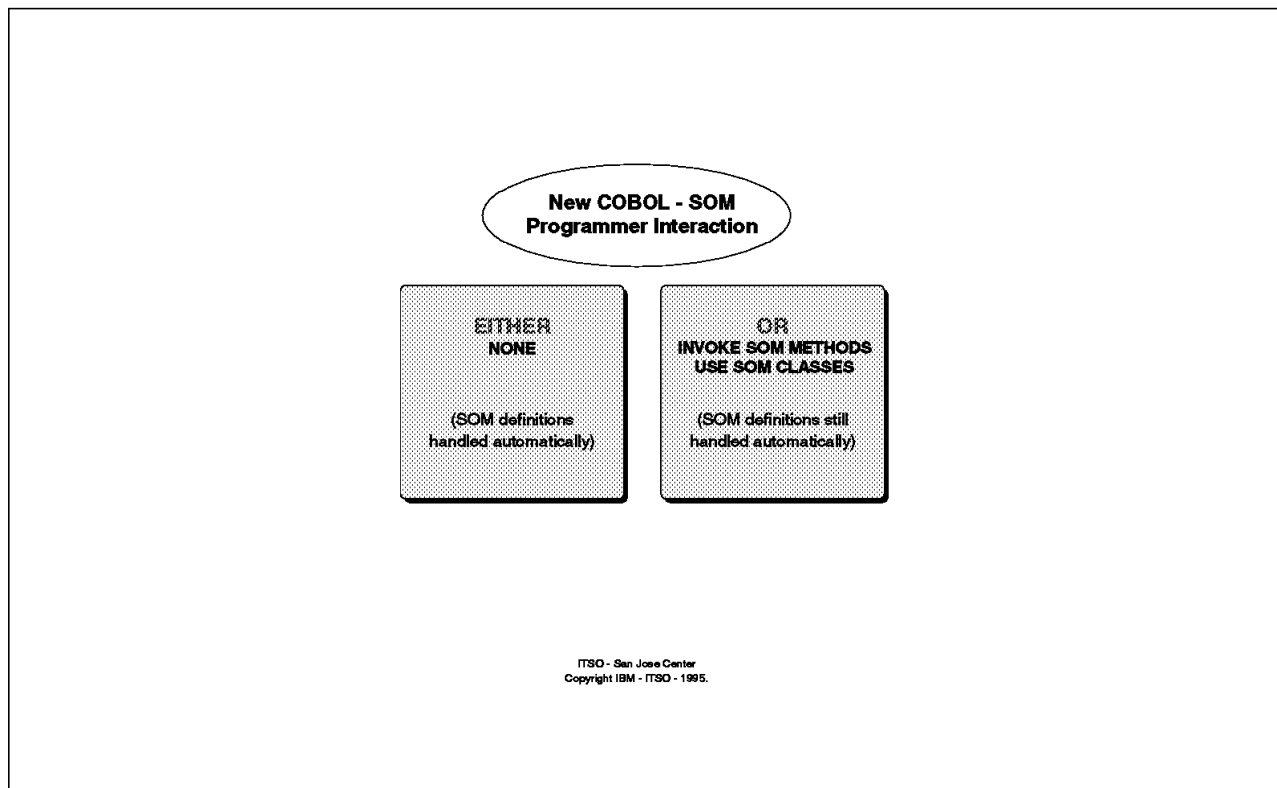


Figure 35. Programmer Use of SOM

However, a programmer can take advantage of a few methods and classes supplied by SOM. For example, SOM provides list capabilities which the programmer could use, rather than creating his own.

SOM therefore provides for COBOL a standard approach, one which ensures maximum portability, flexibility, and interoperability. It moves OO programming from the isolated, single-system approach towards the multi-system, multi-connected object world of the future as shown in Figure 36.

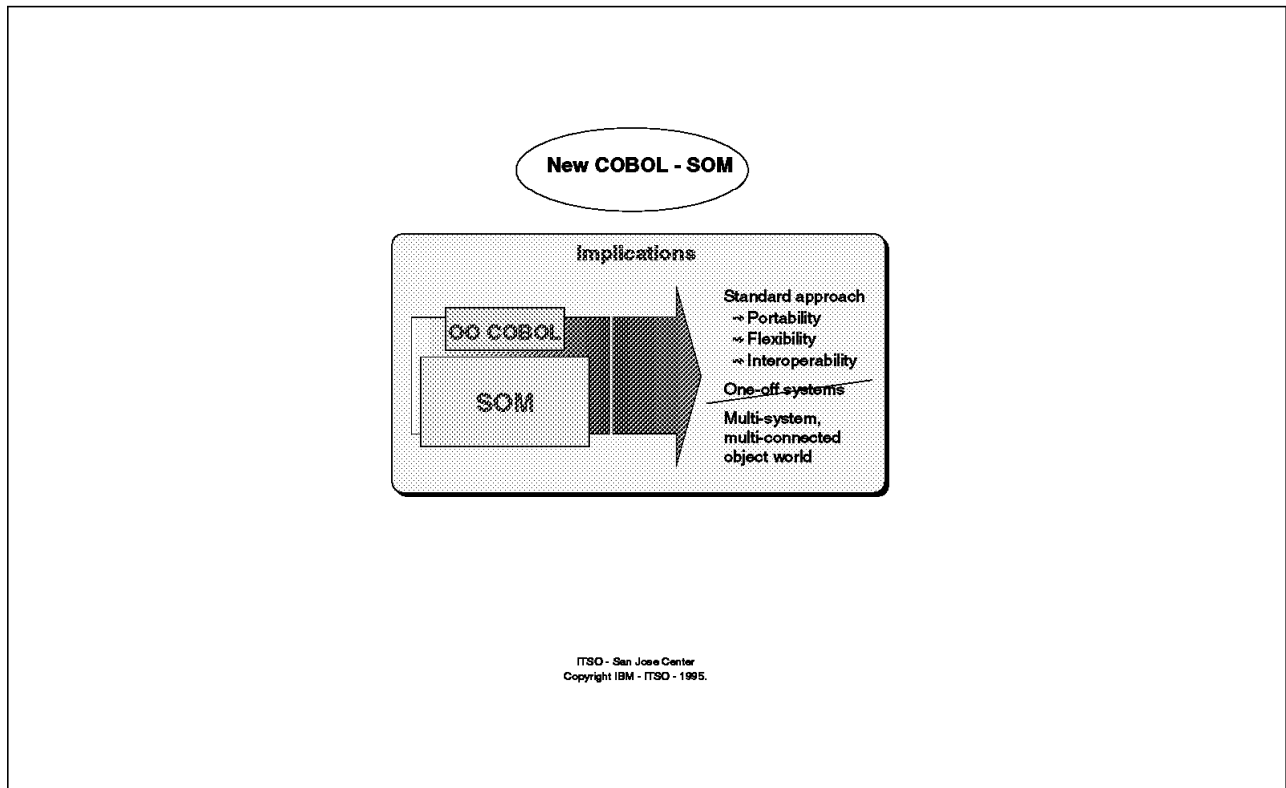


Figure 36. Implications

Because of COBOL's "Direct-to-SOM" facility, COBOL shops can now start the journey, secure in the thought that they will not have to change before completion.

---

## 4.7 Class Libraries

Many class libraries already exist in C++. As they become SOM-enabled they will become available to COBOL programmers.

---

## 4.8 Comparison with Other Languages

IBM provides a complete range of AD offerings. Figure 37 shows the relationship of the products. In the current PL/I product there is no object-oriented capability, while at the other end, it is impossible to use Smalltalk without doing everything in an object-oriented manner.

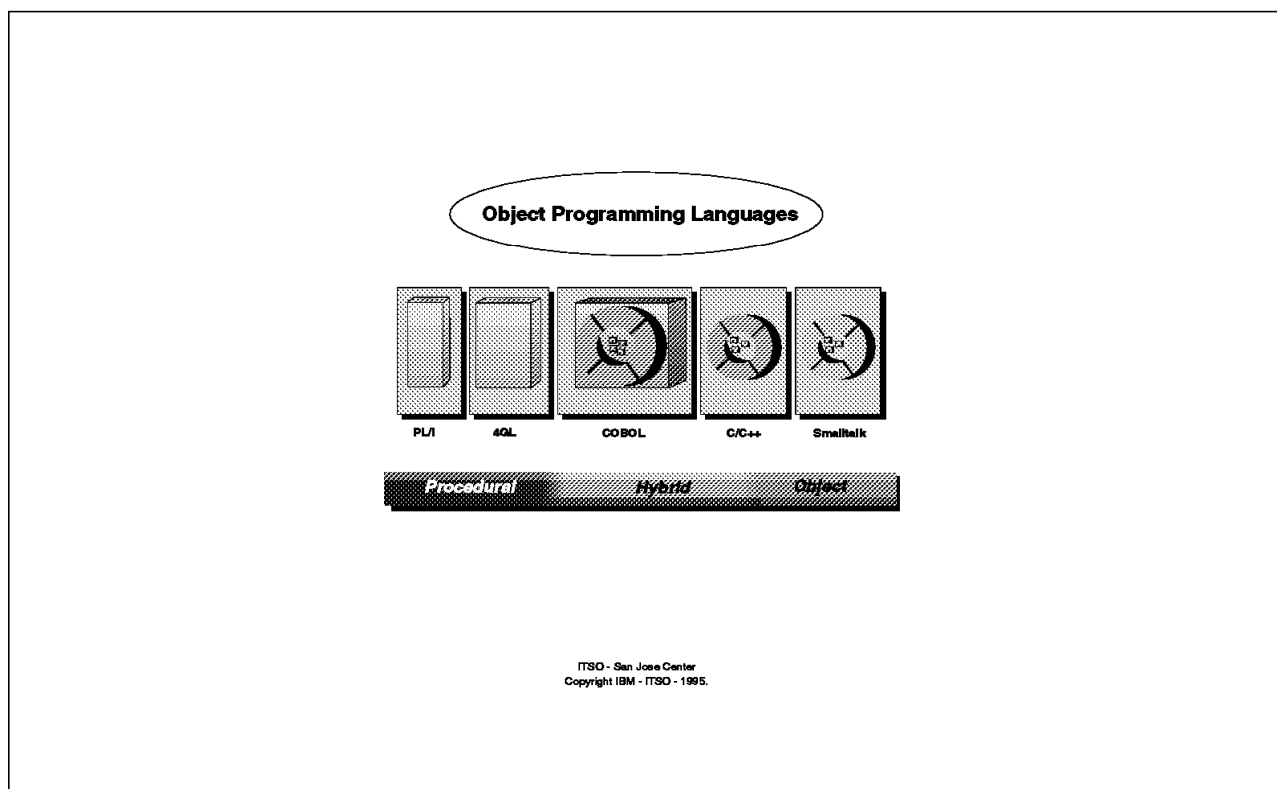


Figure 37. OO Languages

It is possible to use both C++ and COBOL in a procedural way or an object-oriented way. However, it is still not possible to add all the object-oriented constructs into the COBOL product until further progress has been made on the ANSI standard.

As shown in Figure 38, the three leading OO languages are COBOL, C++, and Smalltalk. Many alternative object-oriented implementation decisions have been made.

C++ and Smalltalk also maintain relative and absolute popularity based on the number of skilled practitioners.



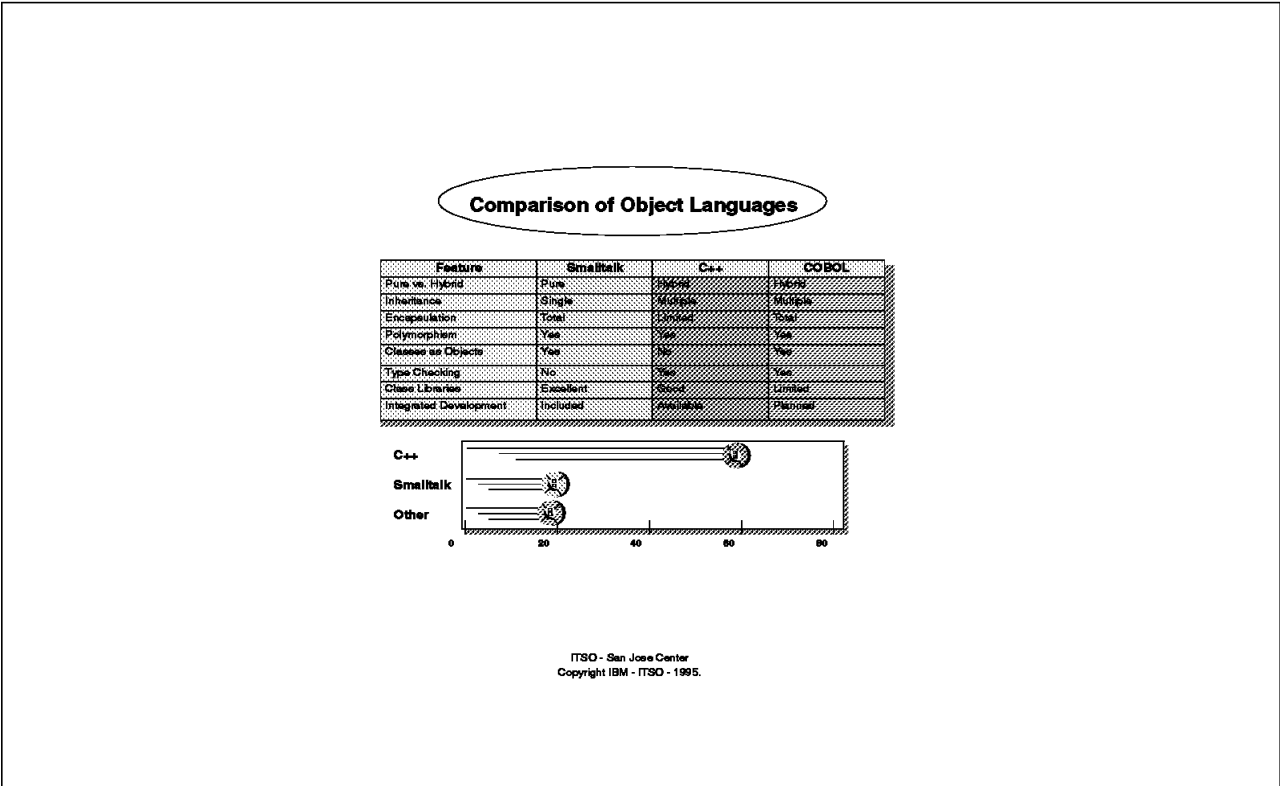


Figure 38. Comparison of OO languages

### 4.9 Further Development

Although COBOL can be described as having an object-oriented capability, further object-oriented facilities need to be added. Object-oriented programming provides a spectrum of capabilities. Requirements that still need to be developed are shown in Figure 39.

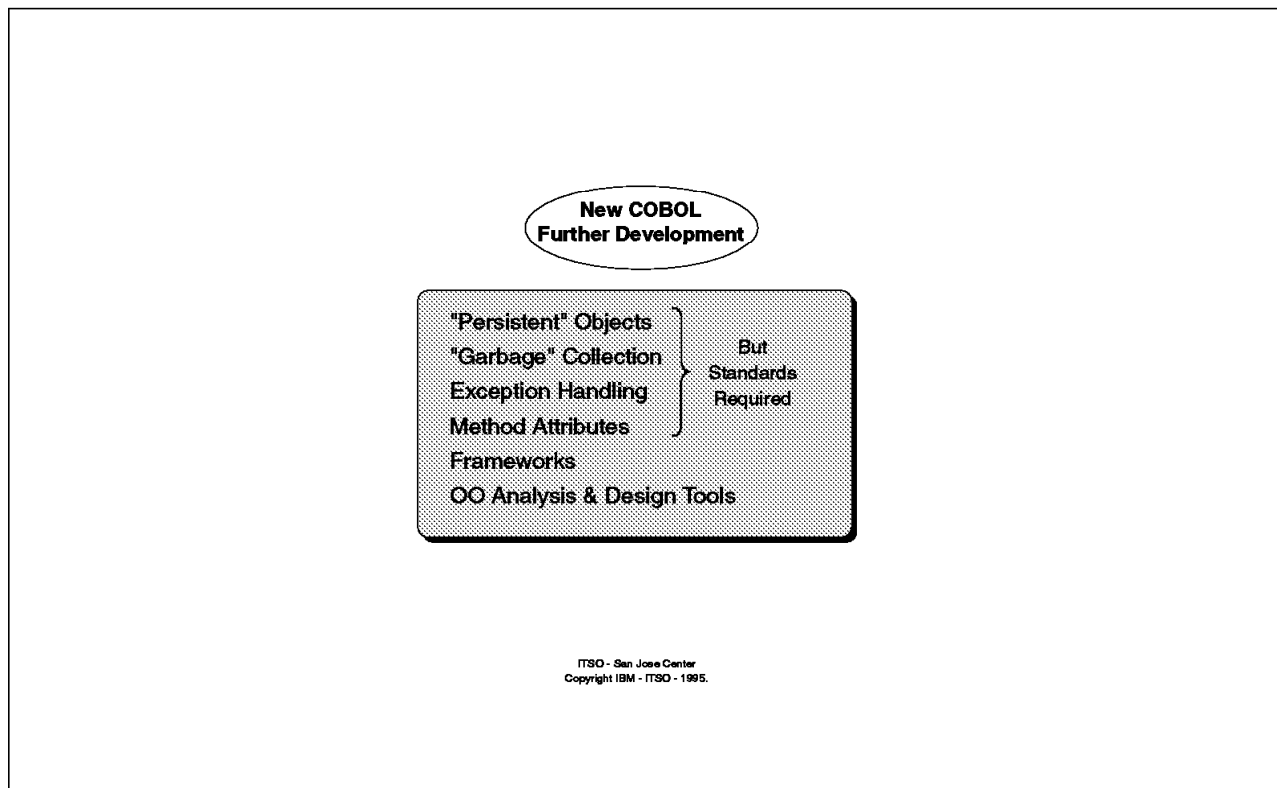


Figure 39. Further Requirements

One feature which might be of value would be what are called persistent objects. Without this, objects exist from when they are invoked until the end of the invoking program. As the name implies, "persistent" objects can continue existing beyond the life of the invoking program, thus allowing other programs and objects to communicate with them.

Associated with that is the need for something called "garbage collection." This is a procedure to clean up any unwanted objects, that is, nonpersistent objects which have not been deleted. Without this clean-up, storage would gradually become taken by all the unused objects.

IBM is interested in providing a robust, industrial-strength solution rather than a toy. As such, it is highly desirable to augment the exception-handling function.

One other refinement would be to specify different categories of use for methods. Currently, anyone can use any method. It might be useful in some circumstances to restrict the use of a method to the object itself.

The absence of an agreed-upon standard inhibits the introduction of extra facilities. This should not be a problem, however, since there is a wealth of object-oriented function in the new product.

## Chapter 5. Implementation

This chapter presents a high level approach to implementing the object-oriented features of COBOL.

### 5.1 Evolution

Implementation offers four logical possibilities: a non-COBOL/370 customer can use OO COBOL on a mainframe or on a workstation, and so can a COBOL/370 customer. This is illustrated in Figure 40.

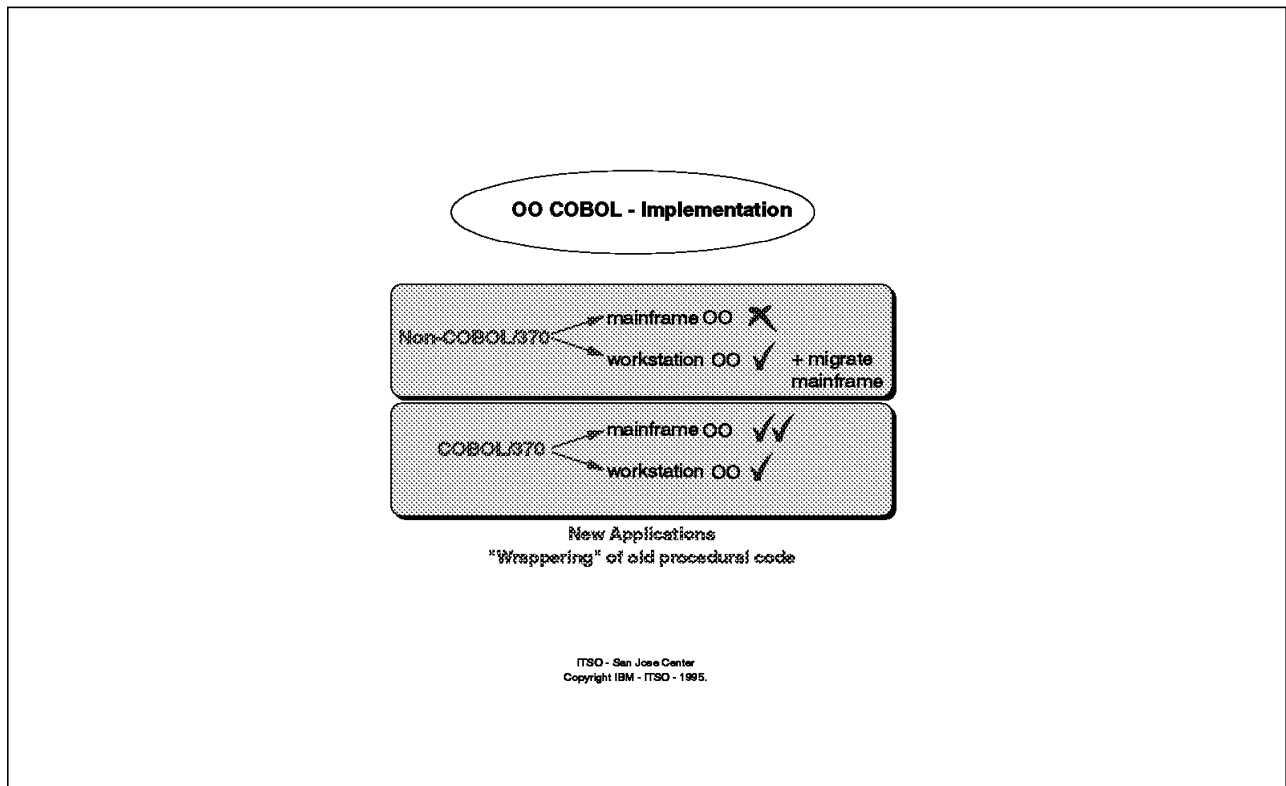


Figure 40. Implementation

Of these four options, the non-COBOL/370 customer should probably avoid the mainframe OO initially because that involves two new concepts simultaneously (OO and Language Environment (LE)). It would be better to work with OO on the workstation and, as a separate exercise, move to LE on the host.

Either option, workstation OO or host OO, is viable for the COBOL/370 user. The host option looks more attractive because there is less new setup. If a team were to be set up to work with OO on the workstation they would probably include a majority of mainframe-experienced people. This might introduce an extra delay in the learning process.

In mainframe migration, the language is a superset of COBOL for MVS & VM V1R1 (known as COBOL/370). Anyone transferring from that level should experience no migration difficulties other than the new Reserved-Word table. This table has all the new object related words added (such as METHOD and INVOKE).

Another consideration is how to interface the new object-oriented capabilities with the existing procedural code. The technique employed here is called “wrapping.” It involves creating an OO shell, or wrapper, around an existing section of procedural code. Thereafter, that procedural code is accessed only via the object-oriented shell.

## 5.2 Dangers and Difficulties

It would be misleading to suggest that the extensive possibilities of this rich new environment can be realized simply by upgrading the installation’s compiler. The new language is easy to learn but to exploit it requires a different way of thinking as well. This is illustrated in Figure 41.

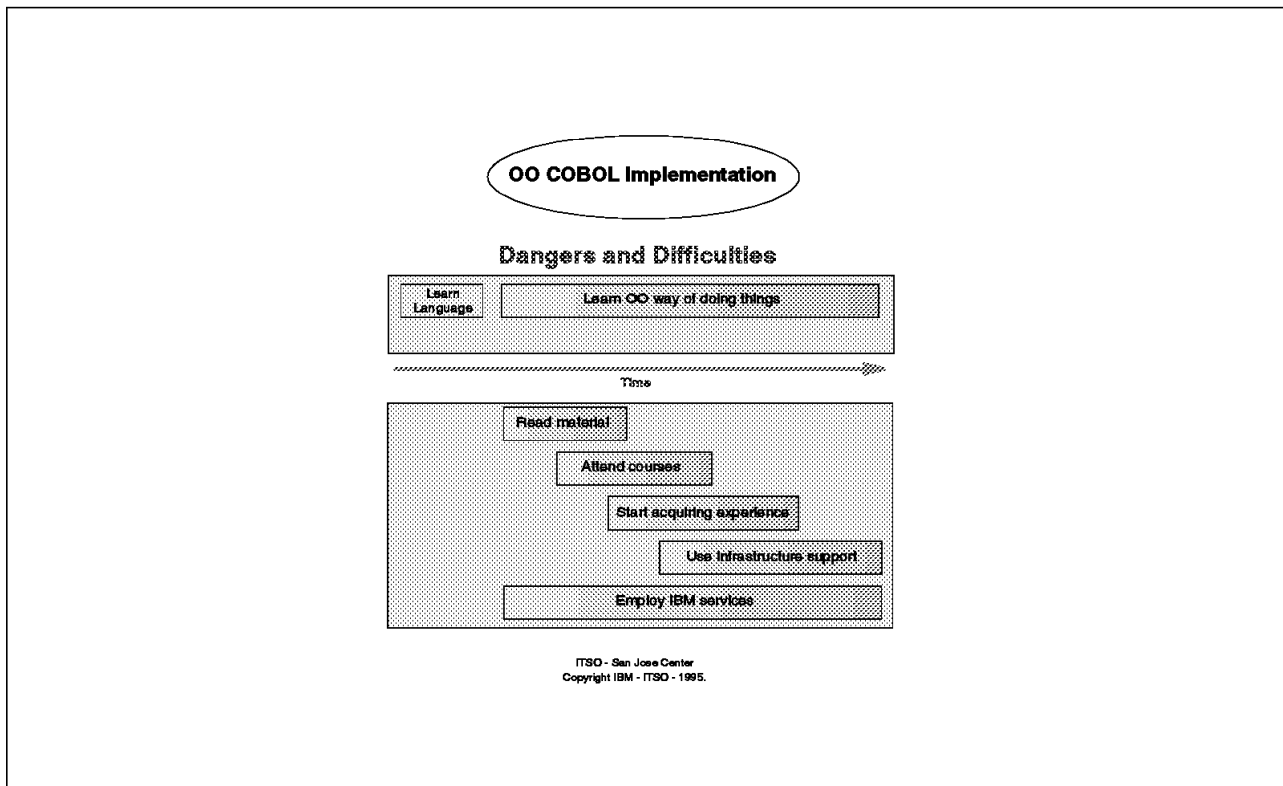


Figure 41. Dangers and Difficulties

It is rather like giving a person who manually typed and mailed letters a PC with E-mail links. If that person uses the PC to type letters, prints them and mails them, not a great deal of benefit has been gained.

The key requirement is the OO design. Applications need to be structured in an object-oriented manner. The use of some of the syntax can only be appreciated with education and experience.

There will also be explicit support for the infrastructure products such as DB2, IMS, and CICS, but (from an OO point of view) that support does not yet exist. So, even if an organization were to have all the skills and experience they would still be forced to develop everything the hard way. See Figure 42.

The learning method is clear:

1. Read available material.

2. Attend appropriate courses.
3. Employ IBM services to fill gaps, to provide mentoring support, and to enhance the education process.
4. Start building experience.
5. Move to using infrastructure support and further OO facilities as they become available.

All of the following references are useful:

IBM Product Manuals  
 IBM Redbooks  
 OO Books  
 OO Forums  
 OO Conference Proceedings

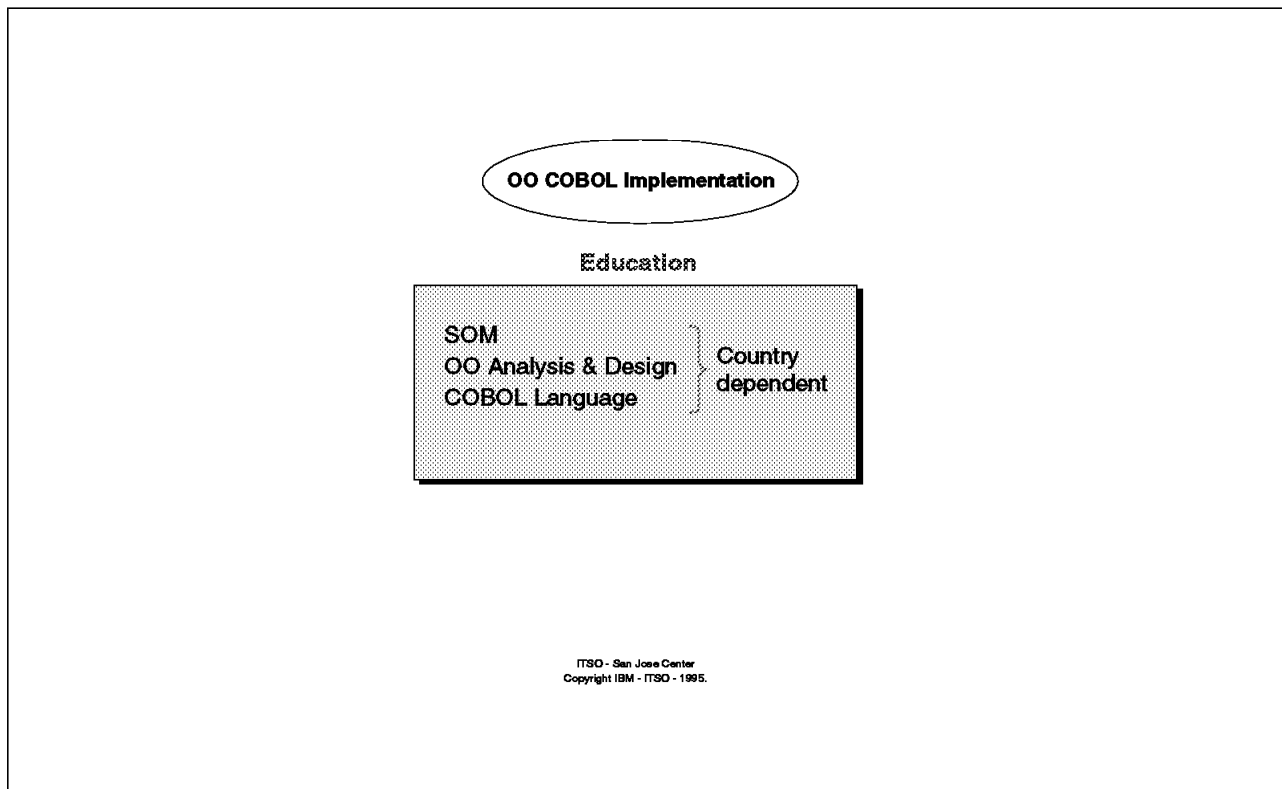


Figure 42. Education

Education will vary from country to country, but should include some or all of the following:

- SOM
- OO Analysis and Design
- COBOL language

### 5.3 IBM Internal Resources

A summary of object-oriented resources provided internally by IBM is shown in Figure 43.

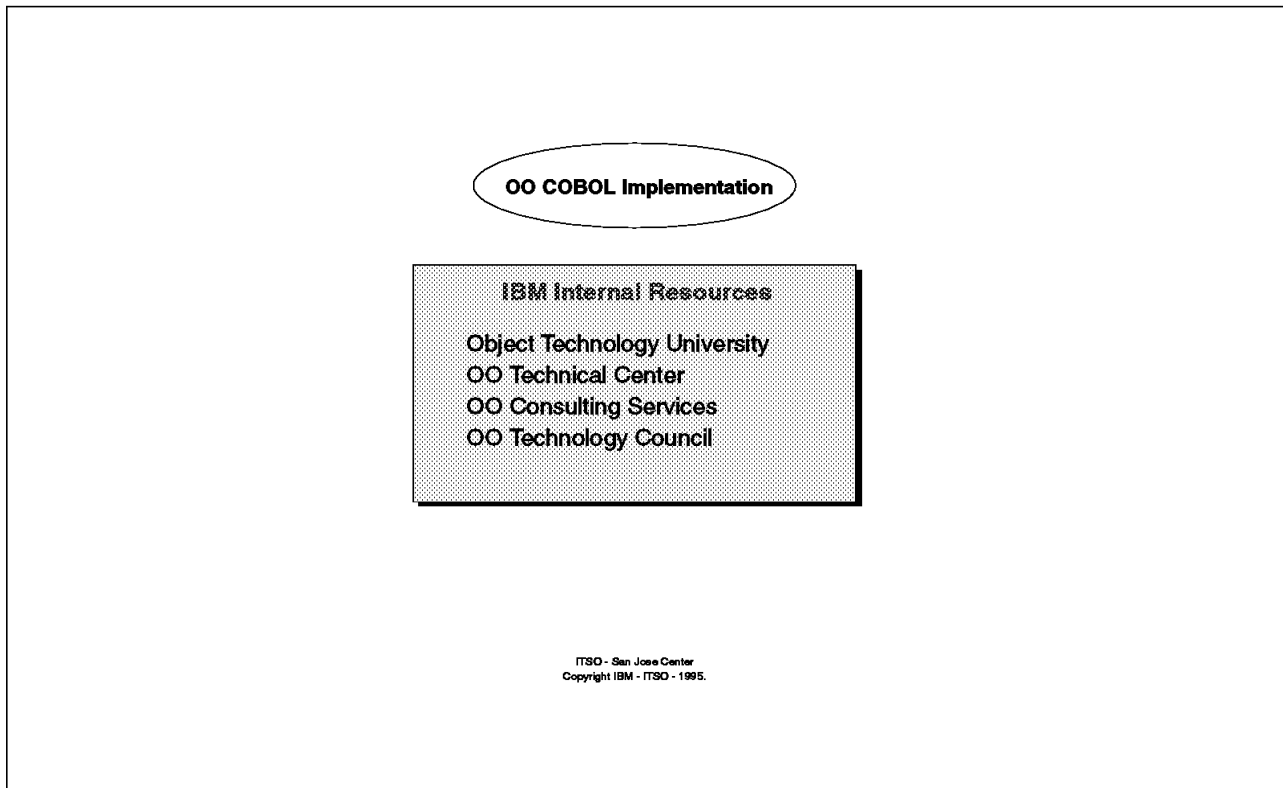


Figure 43. Internal IBM Resources

IBM has built a vast pool of experience in object-oriented technology.

For example, the OO Technology University provides Residency, Extension, Continuing Education, and Special Events Programs. The cornerstone of Object-Technology University (OTU) is the Residency Program. It is an intensive mix of classroom training, case studies, lab exercises, mentoring programs and on-the-job training.

The IBM OO Technology Center (OTC) formed in 1992 supports the move of IBM's software development groups towards the use of object technology. Its tasks include:

- Promotion of OO technology benefits
- Building of expertise
- Development of a document library
- Consulting services
- Assistance to software groups in application assessment and deployment plan preparation

Finally, there is the object-oriented part of the IBM Consulting Services group in Boulder, Colorado.

---

## 5.4 Timescales

It is not possible to quote from COBOL experience since OO technology is so new. However, the Smalltalk example in Figure 44 shows that the elapsed time for building an effective and experienced OO team is years, not months, and that acquiring the language skills is not a major barrier.

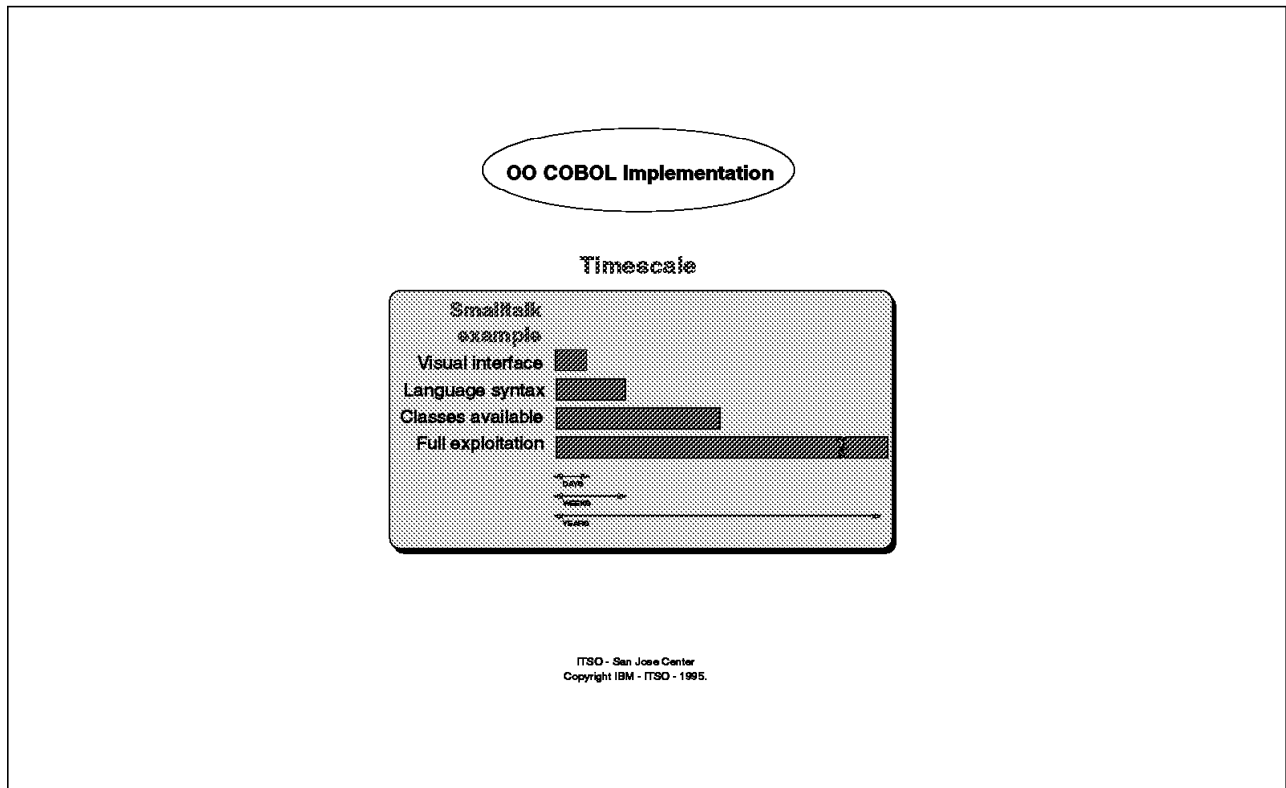


Figure 44. Timescale

## 5.5 Conclusion

IBM's OO COBOL is a technology breakthrough. Bringing together two of the major AD technologies – the one vastly widespread and familiar, the other vastly promising – is itself of major significance. But to do this on both workstation and mainframe platforms, and to do so on the back of CORBA compliant SOM marks a massive milestone in the history of AD as indicated in Figure 45.

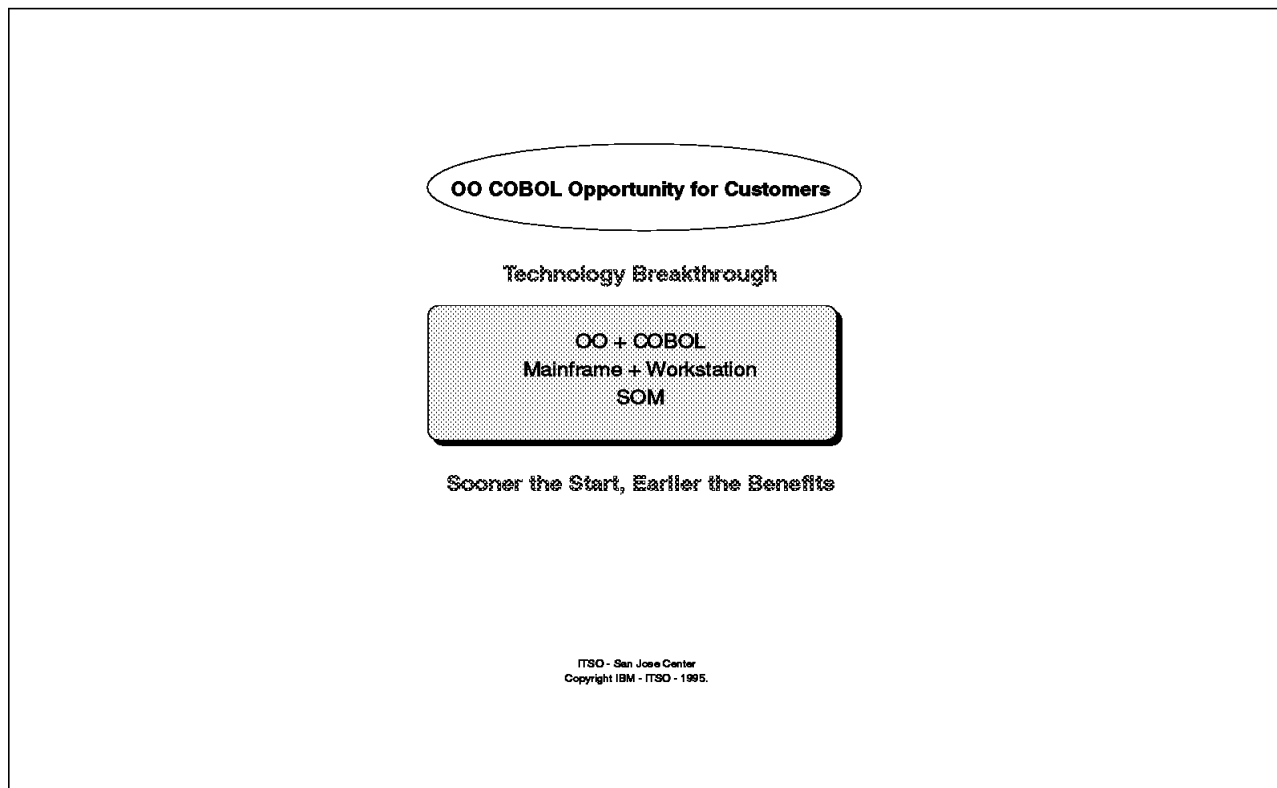


Figure 45. Breakthrough

With mainframe and COBOL support, IBM customers can now take advantage of it object-oriented programming. This will take time. But starting now means customers can develop their competence without the pressures of undue haste. IBM has much to offer in this process that no one else can.



---

## Part 2. Coding Object-Oriented COBOL



---

## Chapter 6. Overview

This chapter contains an overview of object-oriented COBOL programming.

---

### 6.1 Purpose

After completing this section, the reader should be able to write simple object-oriented programs using the new COBOL family. They may be simple ones, but all the new syntax is explained with examples and explanations.

It is assumed that the reader has:

- Basic object-oriented concepts awareness
- Traditional COBOL programming skills

No attempt is made to teach OO design. Nor are any other non-language matters covered, such as compiler options or prelinking requirements.

The reader should understand the fundamental ideas behind object-oriented computing and some of the terminology such as class, method, and inheritance. Terms such as metaclasses and constructors are explained.

This manual uses an imaginary business example to help understand the concepts and syntax being introduced. The running example used refers to the ordering of wine. The only item of knowledge needed is that "claret" is a type of wine.

---

### 6.2 Format

The sequence of topics is:

- Class definition
- Method definition
- Client definition
- Subclasses
- Metaclasses
- SOM use

Within most topics the following sequence is adopted:

1. OO concept review
2. Syntax description
3. Syntax explanation
4. Syntax example
5. Business application

The idea is to build up logically, starting with classes and methods, then the client programs using the methods to access the classes, then the refinement of subclasses, before talking about metaclasses. At the end we'll explain SOM and where to find out more information.

---

## 6.3 Business Problem

The Business Example used as a basis for illustrating the COBOL syntax is a wine retail outlet where, in summary,

- Customers order by the case
- Cases contain 1 to 12 bottles
- System must be able to:
  - Check old case order
  - Create new case order
- Case number, date, and contents stored.

Customers order by the case of 12, which can be mixed and incomplete. In dealing with customers you might want to check an old case order or to create a new case order.

When checking an old case, the customer gives the case number and then you check to see if each bottle is in stock. You produce a list of any that are not in stock. We do this for as many cases as we like and when we stop we want to know how many cases we have checked.

When creating a new case we add bottles, or delete any added by mistake. Once complete, we are given a case number and the cost.

We build this system in three examples. Examples one and two implement part of the final design requirement. The third example implements the overall design requirement.

The first topic considers classes.

---

## 6.4 Code Creation and Processing

There are two types of OO COBOL programs:

- Client programs
- Class programs

Typically, and certainly for the examples described in this manual, there is one client program and several class programs. In this context, the client code is a procedural piece of code which references the class programs as its logical flow demands, just as a conventional program references called programs. These programs can be developed using the Visual Development Environment, but for simplicity, the command-line is used.

The command to invoke the compiler is COB2. The format is as follows:

```
COB2 client1.cb1 class1.cb1 class2.cb1 ... classn.cb1
```

While the class programs can be in any order, the client program must always be named first, since this determines the executable program. The COB2 command compiles all the programs and then, if the compilations have been successful, links all the programs into the one file.

Result from Compile:

```
client1.exe
```

If a compilation fails, then after editing the associated source file, the COB2 command can be issued again. This time it should specify ".obj" for the files that successfully compiled the first time. For example, if after issuing the command

```
COB2 client1.cbl class1.cbl class2.cbl class3.cbl
```

If class1 and class3 failed to compile then ... correct code and use this command class1 and class3 compilations failed while client1 and class2 were successful, it would be necessary to edit class1.cbl and class3.cbl. Then, issue the COB2 command as follows:

```
COB2 client1.obj class1.cbl class2.obj class3.cbl
```

The client program is then invoked as normal by entering

```
client1
```



---

## Chapter 7. Classes

This chapter defines the concept of classes in object-oriented COBOL. Some initial definitions of objects and classes are as follows:

- Objects = data + methods
- Classes define objects
- Example

Class: "US President"

Objects: "Washington," ... "Lincoln," ... "Clinton"

One class, 42 objects.

Data: Name, age-on-election, ...

Methods: "GiveName," "CountVetoed," ...

The first part of this book provided an initial understanding of OO concepts. Just as a reminder, though, objects in OO are combinations of data and the functions that work on that data (called "methods"). The data is only accessible via those methods.

Objects are defined in *classes*. So if there is a US President class, the objects derived from that class would include Washington, Lincoln, two Roosevelts, and a Clinton. There is just the one class but, at the time of writing, 42 objects.

In our business example, the two classes with their data and methods appear as shown in Tables 1 and 2.

<i>Table 1. Winecase Class with Data and Methods</i>	
<b>Winecase</b>	
Data	(Case-number)
	(Case-date)
	(Case-count)
	(Case-contents)
Methods	1: somDefaultInit (override)
	2: AddBottle
	3: RemoveBottle
	4: CalculateCost
	5: GetCaseNumber
	6: DescribeCase

<i>Table 2. Userint Class with Data and Methods</i>	
<b>UserInt</b>	
Data	(User-action)
	(User-bottle)
Methods	1: ReadInput
	2: WriteMessage
	3: WriteOutput

The business problem task is to define two classes : Winecase and Userint.

The wine case has have four items of data with self-explanatory names and six methods whose purpose is as follows.

- 1: somDefaultInit (override)** This method is invoked automatically whenever any object is created or "instantiated". We take advantage of that to initialize our data items here.
- 2: AddBottle** Adds a bottle to our case, setting a flag to indicate success or failure.
- 3: RemoveBottle** Removes a bottle from our case, setting a flag to indicate success or failure.
- 4: CalculateCost** Computes the cost of our case.
- 5: GetCaseNumber** Provides the number of the case, established by the first method.
- 6: DescribeCase** Writes the details of the case to a file, referenced by variable "casedata."

Our other class, UserInt, has just two items of data: user-action and user-bottle. At any one time, the flow of data from the user to the application (by definition, via this user interface class) only contains two items of data: what the user wants to do (the action, for example add a bottle or end the application) and, for the add and delete actions, the bottle details.



The three methods are as follows:

- |                        |                                                                                                                                                                             |
|------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>1: ReadInput</b>    | This asks the user whether he wants to add a bottle, delete a bottle, or end the application. If add or delete is selected, it further asks for the bottle details.         |
| <b>2: WriteMessage</b> | After the actual adding or deleting of a bottle (performed by the appropriate method in "winecase") this method tells the user whether the operation was successful or not. |
| <b>3: WriteOutput</b>  | When all the bottles have been added and deleted, this method tells the user what the cost is and what the case number was.                                                 |

---

## 7.1 Class Definition Structure

COBOL programs defining classes have the following general structure:

- ID Division.  
    Class-Id. Classname ...  
    .....
- Environment Division.  
    Repository.  
        Class Classname ...  
    .....
- Data Division.  
    .....
- Procedure Division  
    Method 1  
    .....  
    Method n  
    .....
- End Class Classname

A class definition is similar to a COBOL program with the following four divisions:

- Identification Division
- Environment Division
- Data Division
- Procedure Division

At the beginning and at the end of the definition, you have a class name instead of a program name. You also mention the classname in the Environment Division. This connects the class name in COBOL to the name in the outside world using SOM. An entry (File Control) in the Environment Division connects the file as COBOL refers to it in the program and the file name as the outside world sees it. The Data Division contains the definitions of all the data items that objects derived from this class use. The Procedure Division contains all the

method definitions, as we'll see in the next chapter. Case matters only within quotes. The period in Repository section is only at the end.

---

## 7.2 Class Definition Statements

The new definition statements look in summary as follows:

- ID Division.  
Class-ID. *Classname* Inherits SOMObject.  
(or)  
Class-ID *Classname* Inherits *SuperClassname*  
.....
- Environment Division.  
Configuration Section.  
Repository.  
Class *Classname* is "*Classname*"  
Class SOMObject is "SOMObject."  
.....
- Data Division.  
Working-Storage Section
- Procedure Division  
... (method definitions, discussed in the next chapter)
- End Class *Classname*.

The words in italics are the words under the user's control. Other words, such as "Class-ID," are part of COBOL.

- Class-ID paragraph.  
In the Class-ID statement, the classname must be the same as the subsequent other two references, in the Repository section and in the End Class statement. Case is not important nor does the name used in this definition have to be the same as references in other definitions or client programs. However, to avoid confusion, be consistent in naming and in case usage everywhere.

The classname after the "inherits" is the class from which this class directly inherits. Every class eventually inherits from SOMObject but only the immediate ancestor is required.

- Repository paragraph  
This is our link to the outside world. Our class definitions held in the SOM Interface Directory where they are referenced by other class definitions and by client programs (in both cases, possibly by non-COBOL code), both at compile and at runtime.

The first name must match our references within our program while the second name, inside quotes, determines what is stored or what is being referenced in the SOM Interface Repository.

**Note:** The quotes are single or double depending on the QUOTE/APOST compiler option setting.

This mechanism links all programs and definitions. The name within quotes is the same for everyone while the first name is local to this definition. Be consistent and use the actual class name everywhere.

List any class the program references directly. If our methods reference other classes, we must quote those too.

- Omissions

There are a number of items which are illegal to define because they do not make sense. For example, there is no Input-Output section. If this is required it is defined in the appropriate method definition.

The Data Division does not allow Local-storage, a new concept, described later. There is no linkage-section. If relevant, it is be defined for the corresponding method.

A less obvious omission is the prohibition of Value clauses. If any flags are used, their level 88 Value settings can still be specified.

While case is not important almost everywhere else, it is important within the quotes on the Class-ID statement. The Repository Section consists of a number of "Class X is 'Y' " statements. Only the last one requires a period (or full stop).



---

## Chapter 8. Methods

This chapter explains methods as used in object-oriented COBOL programming.

Our business example has two classes with nine methods between them as shown in Tables 3 and 4.

<i>Table 3. Winecase Class with Data and Methods – Review</i>	
<b>Winecase</b>	
Data	(Case-number)
	(Case-date)
	(Case-count)
	(Case-contents)
Methods	1: somDefaultInit (override)
	2: AddBottle
	3: RemoveBottle
	4: CalculateCost
	5: GetCaseNumber
	6: DescribeCase

<i>Table 4. Userint Class with Data and Methods – Review</i>	
<b>UserInt</b>	
Data	User-action)
	(User-bottle)
Methods	1: ReadInput
	2: WriteMessage
	3: WriteOutput

The methods are defined in the Procedure Definition of the Class definition.

---

### 8.1 Method Definition Structure

- Procedure Division. (of class definition)
  - ID Division.  
Method-ID. *Methodname* ...  
.....
  - Environment Division.  
.....
  - Data Division.  
.....
  - Procedure Division.  
.....

- End Method *Methodname*.

Like a class definition, a method definition is similar to a COBOL program with the following four divisions:

1. Identification Division
2. Environment Division
3. Data Division
4. Procedure Division

Like a class definition, at the beginning and at the end of the definition, there is the method name instead of the program name.

### 8.1.1 Method Definition Statements

- ID Division.  
Method-ID. *Methodname*.  
.....
- Environment Division.  
Input-Output Section.  
(No Configuration Section.)  
.....
- Data Division.  
File Section.  
Working-Storage Section.  
Local-Storage Section.  
Linkage-Storage Section.
- Procedure Division.  
...
- End Method *methodname*.

The words in italics are the words under user's control. The other words, such as Method-ID, are part of COBOL.

- Method-ID paragraph.

In the Method-ID statement, the *methodname* must be the same as in the End Method statement. Case is not important.

An optional keyword *Override* is specified if the method name is the same as a method defined further up the inheritance chain. We shall discuss this further in the sub-class topic.

- Environment Division

There is no a configuration section. The File Section is here just as in a normal program.

- Data Division

The difference between Local-Storage and Working-Storage is that data in working storage is always accessed in the last-used state when the method is invoked. Conversely, data in local-storage is refreshed every time the method is called.

Unlike the Working-Storage data defined at the class level, the Value clause can be used, both for working-storage and for local-storage.

- Procedure Division

The only new statement unique to the method definition is Exit Method. This is analogous to the Exit Program statement in a program. The statement Goback can be substituted, as it can be for Exit Program in programs.

**Note:** There is a new statement (Invoke). New variants of old statements can be specified here but we shall come across them in the next section, dealing with client programs.





---

## Chapter 9. Client

This chapter explains the concept of the client in object-oriented COBOL programming.

---

### 9.1 Client Program Example

Tables 5, 6, and 7 illustrates the logic of our sample program "Wine."

Table 5. Client Program Logic Flow Part One	
1	Create 'Userint' Object
2	Create 'Case' Object

Table 6. Client Program Logic Flow Part Two			
3	Ask Add/Delete/End (U)	4	Do Add/delete (C)
		5	Write message (U)
		6	Repeat question 3 (U)

Table 7. Client Program Logic Flow Part Three	
7	Calculate cost (C)
8	Find case-number (C)
9	Tell user (U)
10	Free objects (C) (U)

Between the creation and freeing of the objects, there is a simple loop asking the user to add or delete bottles, followed by a short calculation and a message issued to the user.

The letters (C) and (U) indicate object references to the wineCase and Userinterface objects, respectively.

---

### 9.2 Client Definition Structure

The client program is similar in structure to any conventional program.

- ID Division.  
Program-ID. *Programname* ...  
.....
- Environment Division.  
.....
- Data Division.  
.....
- Procedure Division.  
.....
- End Program *Programname*

A client program is an ordinary program with the same structure. However, there are some extra statements used. These are described in the next section.

The words in italics are the words under the user's control. The other words, such as Program-ID, are part of COBOL.

---

## 9.3 Client Definition Statements

The details of the new definition statements are as follows:

- Environment Division.
  - Configuration Section.
  - Repository.
  - Class *Classname* is "*Classname*".
  - .....
- Data Division.
  - Working-Storage Section.
  - 01 *objecthandle1* Usage Object Reference.
  - 01 *objecthandle2* Usage Object Reference *Classname*.
- Procedure Division.
  - ... (described later)
- End Program *Programname*.
- Repository paragraph.

The rules are similar to the Class definition, that is:

- Every class referenced in the program must be mentioned here.
- There must be a full-stop (period) after the last Class statement, and none elsewhere.
- Case is not important, except between the quotes, where it is essential.

- Data Division

The Object Reference is a new usage type. This allows statements in the procedure division to interact with objects, either by using their methods or by comparing objects.

There are two varieties: one with a class referenced and one not. (There is a third type of definition for metaclasses).

The one referencing a class is called a typed object reference and can only be used with objects instantiated either from that class or from subclasses of that class.

The other object reference is called a universal object reference and can be used with any object.

All class names mentioned here must also be mentioned in the Repository paragraph so that COBOL can look them up in the SOM Interface Repository.

---

## 9.4 Object Reference Statement Details

- Data Division.
  - Working-Storage Section. (or Linkage or Local)
  - 01 *univObj* Usage Object Reference.
  - 01 *caseObj* Usage Object Reference Winecase.
  - 01 *Labels*.
    - 05 *labelname* Pic X(15).
    - 05 *labelObj* Usage Object Reference Label.
- Procedure Division valid verbs (described later)
  - Invoke *caseObj*
  - Set *univObj* to *caseObj*
  - If *univObj* = *caseObj*
  - Invoke ... Using/Returning *caseObj*
  - Call ... Using/Returning *caseObj*
  - Call ... Using/Returning *caseObj*

Object References, although introduced here in client programs, can be defined in any section of the data division of a class, method, or client program.

Object References can be at any level, other than 66 and 88, and as part of a group. (If the object reference is itself the group item, it is the item at the lowest part of the group which represents the actual handles.)

The only way an object reference data item can be used in the procedure division is with the verbs shown here.

---

## 9.5 Interacting with Objects

There are four types of object interaction:

- Creating Objects
  - Invoke *Classname* "somNew" Returning *handle*
- Using Methods
  - Invoke *handle* "methodname" Using ... Returning ...
- Other Object Operations
  - IF *handle* = Null (or Nulls or *handle2*)
  - SET *handle* = Null (or Nulls or *handle2*)
- Freeing Objects
  - Invoke *handle* "somFree"

In Creating Objects it is *Classname* not *handle*.

Creating an object is also known as creating an instance of a class. Do this with the Invoke statement. We invoke the method somNew against the classname asking the system to return us a handle in the specified handlename. Yet we have already tied the classname and the handlename together in the Object Reference statement in the Data Division. Does this give the system unnecessary information?

The handlename might have been specified on a universal Object Reference and so we must specify the sort of object we want.

To use a method, we again use Invoke, specifying both our handlename and the method name. Methodname is typically a literal, although it can be a variable. In this case the handle must relate to a universal object, not a typed object.

If we are passing data to the method, we specify the data item name in the Using clause. If we expect data back from the method, we specify that data item in the returning clause.

**Note:** Returning is now available on the CALL statement too. These using and returning data items must be defined on the corresponding Procedure Division statement of the method being invoked.

We can also perform comparisons and settings. We can compare one handle with another, or check if it has not been set. Similarly, we can set the handle to another or to be as if it has not been set. Typically, universal object references are used in these cases.

We free an object by issuing the somFree method against the handle.

All our interactions reference a handle of the object with the exception of the somNew method, issued against the classname. Until the object has been created we cannot invoke any of its methods. Actually there is an object for each class defined in our Interface Repository (not to be confused with objects that we create from our class's definition) and it is that object whose method somNew we invoke.

---

## Chapter 10. Example One

This chapter contains examples of the code used in the first version of the sample Wine program, illustrating class definition and object interaction. For space reasons, the comment lines have been omitted. The simplicity of the code should facilitate easy comprehension. Appendix A contains the full code listing.

---

### 10.1 WineCase Class Example

```
CLASS-ID. Winecase Inherits SOMObject.
ENVIRONMENT DIVISION.
Configuration Section.
Repository.
    CLASS SOMObject IS "SOMObject"
    CLASS Winecase IS "Winecase".
DATA DIVISION.
Working-Storage Section.
01 Case-Number      Pic 9(5).
01 Case-date        Pic X(8).
01 Case-Count       Pic 99.
01 Case-Contents.
    05 Case-Entry occurs 12 times.
        10 Case-Bottle Pic X(20).
PROCEDURE DIVISION.
*
*
*
IDENTIFICATION DIVISION.
METHOD-ID. "somDefaultInit" OVERRIDE.
PROCEDURE DIVISION.
    Compute Case-number = Function Random (99999)
    Move "01011996" to Case-date
    Move 0 to Case-count
    Initialize Case-Contents.
    Exit Method.
END METHOD "somDefaultInit".
```

```

IDENTIFICATION DIVISION.
METHOD-ID. "AddBott".
DATA DIVISION.
Working-Storage Section.
77  sub   Pic 99 VALUE 0.
01  Found-Flag   Pic 9.
    88  found      VALUE 0.
    88  not-found  VALUE 1.
Linkage Section.
01  In-bottle   Pic X(20).
01  Add-flag    Pic 9.
PROCEDURE DIVISION USING In-bottle Returning Add-flag.
    Set not-found to True
    Move 1 to Add-flag
    Perform varying sub from 1 by 1
        until (sub > 12) or (found)
        IF Case-Bottle(sub) = SPACES
            Move in-bottle to Case-Bottle(sub)
            Add 1 to Case-Count
            Move 0 to Add-flag
            Set found to TRUE
        END-IF
    End-Perform.
    Exit method.
END METHOD "AddBott".

```

```

IDENTIFICATION DIVISION.
METHOD-ID. "RemoveBott".
DATA DIVISION.
Working-Storage Section.
77  sub    Pic 99 VALUE 0.
01  Found-Flag    Pic 9.
    88  found      VALUE 0.
    88  not-found  VALUE 1.
Linkage Section.
01  Out-bottle    Pic X(20).
01  Delete-flag   Pic 9.
PROCEDURE DIVISION USING Out-bottle Returning Delete-flag.
    Set not-found to True
    Move 1 to Delete-flag
    Perform varying sub from 1 by 1
        until (sub > 12) or (found)
        IF Case-Bottle(sub) = Out-bottle
            Move SPACES to Case-Bottle(sub)
            Subtract 1 from Case-Count
            Move 0 to Delete-flag
            Set found to TRUE
        END-IF
    End-Perform.
    Exit method.
END METHOD "RemoveBott".

```

```

IDENTIFICATION DIVISION.
METHOD-ID. "CalculateCost".
DATA DIVISION.
Working-Storage Section.
77  sub   Pic 99 VALUE 0.
77  cost  Pic 9(5)V99.
Linkage Section.
01  Total-cost      Pic 9(7)V99.
PROCEDURE DIVISION Returning Total-cost.
    Move 0 to Total-cost
    Perform varying sub from 1 by 1
        until sub > case-count
        ADD 1    to Total-cost
    End-Perform.
    Exit method.
END METHOD "CalculateCost".
*
*
IDENTIFICATION DIVISION.
METHOD-ID. "GetCaseNumber".
DATA DIVISION.
Linkage Section.
01  Case-num      Pic 9(5).
PROCEDURE DIVISION Returning Case-num.
    Move Case-number to Case-num.
    Exit method.
END METHOD "GetCaseNumber".

```



```

IDENTIFICATION DIVISION.
METHOD-ID. "DescribeCase".
ENVIRONMENT DIVISION.
Input-Output Section.
File-Control.
    SELECT case-file ASSIGN to CaseData
    File Status is Data-key
    Organization is Line Sequential.
DATA DIVISION.
File Section.
FD case-file External
    Record contains 255.
01 case-record Pic X(255).
Working-Storage Section.
01 Data-key      Pic X(2).
01 print-line.
    05 print-case-number    Pic 9(5).
    05 print-case-date      Pic X(8).
    05 print-case-count     Pic 99.
    05 print-case-contents.
        10 print-case-entry occurs 12 times.
        15 print-case-bottle Pic X(20).
PROCEDURE DIVISION.
    Open Output case-file
    Move case-number  to print-case-number.
    Move case-date    to print-case-date.
    Move case-count   to print-case-count.
    Move case-contents to print-case-contents.
    Write case-record FROM print-line.
    Close case-file.
    Exit method.
END METHOD "DescribeCase".
END CLASS Winecase.

```

## 10.2 UserInterface Class Example

Version 1

CLASS-ID. UserInterface Inherits SOMObject.

ENVIRONMENT DIVISION.

Configuration Section.

Repository.

CLASS SOMObject IS "SOMObject"

CLASS UserInterface IS "UserInt".

DATA DIVISION.

Working-Storage Section.

01 User-action Pic X(10).

88 User-add Value "Addbott".

88 User-delete Value "Deletebott".

88 User-end Value "End".

01 User-Bottle Pic X(20).

PROCEDURE DIVISION.

\*

IDENTIFICATION DIVISION.

METHOD-ID. "ReadInput".

DATA DIVISION.

Linkage Section.

01 Action Pic X(10).

01 Bottle Pic X(20).

PROCEDURE DIVISION Returning Bottle Action.

Display "Enter the action : add, delete, end"

Accept action from SYSIN

Move Function Upper-case(action) to Action

Evaluate action

When "ADD"

Set User-add to TRUE

Perform Get-item

When "DELETE"

Set User-delete to TRUE

Perform Get-item

When "END"

Set User-end to TRUE

End-evaluate

Move User-action to action

Exit Method.

Get-item.

Display "Enter the item"

Accept Bottle from SYSIN

Move Bottle to User-Bottle.

END METHOD "ReadInput".

```

IDENTIFICATION DIVISION.
METHOD-ID. "WriteMessage".
DATA DIVISION.
Working-Storage Section.
01  action    Pic X(10).
01  bottle    Pic X(20).
Linkage Section.
01  Flag      Pic 9.
PROCEDURE DIVISION Using Flag.
    Move user-Action to Action
    Move user-Bottle to Bottle
    IF flag = 0
        Display action " successfully completed on " bottle
    ELSE
        Display action " unsuccessfully completed on " bottle
    END-IF.
    Exit Method.
END METHOD "WriteMessage".
*

IDENTIFICATION DIVISION.
METHOD-ID. "Writeoutput".
DATA DIVISION.
Working-Storage Section.
77  Formatted-cost  Pic $Z,ZZZ,ZZ9.99.
Linkage Section.
01  Total-cost      Pic 9(7)V99.
01  Case-number     Pic 9(5).
PROCEDURE DIVISION Using Total-cost Case-number.
    Move total-cost to Formatted-cost
    Display "Your order costs " Formatted-cost
    Display "Your case number is " Case-number
    Exit Method.
END METHOD "Writeoutput".
END CLASS UserInterface.

```

## 10.3 Wine Client Example

Version 1

```
PROGRAM-ID. Wine.
ENVIRONMENT DIVISION.
Configuration Section.
Repository.
    CLASS SOMObject    IS "SOMObject"
    CLASS Case        IS "Winecase"
    CLASS UserInterface IS "UserInt".
DATA DIVISION.
Working-Storage Section.
77 caseObj          Usage Object Reference Case.
77 userObj          Usage Object Reference UserInterface.
77 Case-number      Pic 9(5).
77 total-cost       Pic 9(7)V99.
77 action           Pic X(10).
77 bottle           Pic X(20).
77 flag             Pic X.
PROCEDURE DIVISION.
    Invoke UserInterface "somNew" RETURNING userObj
    Invoke Case          "somNew" RETURNING caseObj
    Invoke userObj "ReadInput" Returning bottle action
    Perform until action = "End"
        IF action(1:3) = "Add"
            Invoke caseObj "AddBott" Using bottle Returning flag
        ELSE
            Invoke caseObj "DeleteBott" Using bottle Returning flag
        END-IF
        Invoke userObj "WriteMessage" Using flag
        Invoke userObj "ReadInput" Using bottle action
    End-Perform
    Invoke caseObj "CalculateCost" Returning total-cost
    Invoke caseObj "GetCaseNumber" Returning case-number
    Invoke userObj "WriteOutput" Using total-cost case-number
    Invoke caseObj "DescribeCase"
    Invoke caseObj "somFree"
    Invoke userObj "somFree"
    STOP RUN.
END PROGRAM Wine.
```

---

## Chapter 11. Subclasses

This chapter explains the concept of subclasses in object-oriented COBOL programming.

An example of classes and associated methods is as follows:

- Classes
  - Elephant
  - Jellyfish
  - Animal
  - Cat
  - Dog
- Methods
  - Describe-habitat-of
  - Get-length-of-backbone
  - Give-food-to
  - Be-firm-with

When we analyze a particular system, we might establish a list as shown above, together with a list of potential methods. When we look at the list we can see that some methods, such as describe-habitat-of or give-food-to might apply to all the classes. However other methods might not. For example, get-length-of-backbone would not apply to the jellyfish and the method be-firm-with would not apply to a cat.

A similar set of mismatches can be seen with data items. It would be possible to take each class separately and define its methods repeatedly but this would not maximize the reuse.

So the concept of subclasses is needed. In this example, we can define a parent class, Animal, with subclasses Cat, Dog, Elephant, and Jellyfish. We define the animal class as having all the data items that all its subclasses have and all the methods that apply everywhere.

Then the subclasses only need the methods and data relevant to them. Thus when we send a method get-length-of-trunk to the elephant, it deals with it. However if we send describe-habitat to the elephant, the class passes it to the animal class.

---

### 11.1 Business Example

Tables 8, 9, 10, and 11 illustrate our business example.

Table 8 (Page 1 of 2). Winecase Class with Data and Methods (Subclasses)	
Winecase	
Data	(Case-number)
	(Case-date)

<i>Table 8 (Page 2 of 2). Winecase Class with Data and Methods (Subclasses)</i>	
<b>Winecase</b>	
	(Case-count)
	(Case-contents)
Methods	1: somInit (override)
	2: SetInstanceData
	3: GetInstanceData
	4: AddBottle
	5: RemoveBottle
	6: CalculateCost
	7: GetCaseNumber
	8: DescribeCase

<i>Table 9. Data Class (Subclasses)</i>	
<b>Data</b>	
Methods	1:
	...
	9: ReadCase
	10: CheckBott

<i>Table 10. Newcase Class (Subclasses)</i>	
<b>Newcase</b>	
Data	
Methods	1:
	2:
	3:
	4:
	5:
	6:
	7:
	8:

<i>Table 11. Userint Class (Subclasses)</i>	
<b>Userint</b>	
Data	(User-action)
	(User-bottle)
Methods	1: ReadInput
	2: WriteMessage
	3: WriteOutput

In our business example we have two types of cases:

- One which has already been ordered
- One for those to be created

Up to now we have been dealing with the latter type. We assumed that we only want to define new cases. But in fact we want to be able to query existing cases, to see what their cost is and to check whether the bottles specified are in stock. Doing this requires two new methods: ReadCase and CheckBott. These are not new cases so they are defined in the OldCase class.

In COBOL, unlike some other OO programming languages, instance data is private, methods are public. So a subclass has no difficulty in using the methods of its superclass, for example here OldCase can use CalculateCost, but it cannot directly refer to any instance data in the superclass.

So, it is the responsibility of the superclass to provide methods to allow the subclass to access data such as GetInstanceData and SetInstanceData.

---

## 11.2 Subclass Definition Statements

The structure for defining a subclass is identical to the definition of a class as shown earlier. This is because all classes are subclasses. In our previous examples, there were subclasses of SOMObject, as follows:

- ID Division.  
Class-ID. *Classname* Inherits *superclassname*.  
.....
- Environment Division.  
Configuration Section.  
(No Input-Output Section)  
Repository.  
Class *Classname* is "*Classname*"  
Class *Superclassname* is "*Superclassname*".  
.....
- Data Division.  
Working-Storage Section.  
Only EXTRA instance data  
(No Local Storage, NO Linkage)  
(No Value clauses bar level 88 items)
- Procedure Division.
- End Class *Classname*.

The definition of a subclass is very similar to that of a class.

In the Class-ID statement naming the subclass, the class from which it is directly inheriting, generically referred to as a Superclass, must be explicitly named. Both these classes must be named in the Repository section and any classes

referenced by the subclass's methods. Classes only referenced by the superclass do not need to be listed.

**Note:** It is possible in OO COBOL to inherit from more than one other class. This picks up methods from every superclass specified plus their inheritance chains. If this is required, the immediate parent classes must be mentioned on this statement. They must also be referenced in the Repository section.

In the Working-Storage section, it is only the EXTRA data which needs to be defined. Data defined in the superclass is not defined here. As mentioned before, the subclass cannot directly access this data and has to do so like any other class using the superclass's own methods.

---

### 11.3 Method Definition Statements

Methods are often variants of methods already defined in a superclass.

- ID Division.

Method-ID. *Methodname*.

(or)

Method-ID. *Methodname* Override.

.....

Using Overridden method

Method-ID. CalculateCost Override.

.....

Invoke SUPER "CalculateCost" returning Origcost

Calculate Cost = 0.9 \* Origcost

.....

Methods in subclasses are defined in the same way as those in classes. Often, though, they override ones with the same name in the superclass. In these cases the word Override is added to the name.

We have already seen this when `somDefaultInit` was overridden. Every class (except `SOMObject`) has a superclass in IBM OO COBOL. The inheritance chain always leads to `SOMObject`. Thus, in any class we can override, at the very least, the 22 (at the current count) methods defined in `SOMObject`, one of which is this "`somDefaultInit`."

Everything else about the definition is identical to the class definition. There are however some coding aspects of the Procedure Division which are particularly relevant in subclass methods.

The first occurs when we want to use the original superclass method. In these cases the SUPER term is used.

Suppose for our `Oldcase`, we wanted to have an amended `CalculateCost` method, which cut 10% of the cost if the customer had been waiting over a week for his order. To do this, we override the method in our `Oldcase` subclass. But we do not want to rewrite all the good code in the `CalculateCost` method already in



class Winecase. So in the override method, we invoke SUPER CalculateCost and then multiply the result by 0.9.

The system interprets the word SUPER as meaning “look in the parent class for this method” (“and then in the grandparent” and so on). SUPER should only be specified in method code, not in client code.

---

## 11.4 Accessing Data in Superclasses

The phrase “Public Methods, Private Data” means that while it is possible for a class to treat a superclass’s methods as its own, it does not have automatic access to the superclass’s data. Table 12 illustrates the superclass’s data and methods.

<i>Table 12. Winecase Superclass Data and Methods</i>	
<b>Winecase</b>	
Data	(Case-number)
	(Case-date)
	(Case-count)
	(Case-contents)
Methods	1: somlNit (override)
	2: SetInstanceData
	3: GetInstanceData
	4: AddBottle
	5: RemoveBottle
	6: CalculateCost
	7: GetCaseNumber
	8: DescribeCase

- In Oldcase subclass,  
    Method “CheckBott”  
    ....  
    Invoke SELF “GetInstanceData” Returning Case-details  
    ....

In the superclass we write two methods, one to read all the data and one to write it all. From the subclass we issue the Invoke SELF statement specifying the appropriate method name.

“SELF” is shorthand for asking the system to invoke the specified method against the object from which it is being issued (and then against the object’s parent, and so on). SELF, like SUPER, should only be specified in method code, not in client code.

The difference between a subclass invoking this method and a class doing so is that the subclass does not need to instantiate the superclass; it is the object. By contrast, another class invoking GetInstanceData against the winecase class would first have to obtain an actual object representing the actual case.



---

## Chapter 12. Example Two

With new constructs of subclasses, method overrides, and using SUPER and SELF, we can now construct a second version. Appendix B contains the full code listing.

---

### 12.1 Winecase

```
CLASS-ID. Winecase Inherits SOMObject.  
ENVIRONMENT DIVISION.  
Configuration Section.  
Repository.  
    CLASS SOMObject IS "SOMObject"  
    CLASS Winecase IS "Winecase".  
DATA DIVISION.  
Working-Storage Section.  
01 Case-Number      Pic 9(5).  
01 Case-date        Pic X(8).  
01 Case-Count       Pic 99.  
01 Case-Contents.  
    05 Case-Entry occurs 12 times.  
        10 Case-Bottle Pic X(20).  
PROCEDURE DIVISION.  
*  
*  
*
```

```
IDENTIFICATION DIVISION.  
METHOD-ID. "somDefaultInit" OVERRIDE.  
Same as Version 1  
END METHOD "somDefaultInit".
```

```

IDENTIFICATION DIVISION.
METHOD-ID. "SetInstanceData".
DATA DIVISION.
Linkage Section.
01 In-case.
    05 in-case-number    Pic 9(5).
    05 in-case-date      Pic X(8).
    05 in-case-count     Pic 99.
    05 in-case-contents.
        10 in-case-entry occurs 12 times.
            15 in-case-bottle Pic X(20).
PROCEDURE DIVISION USING In-case.
    Move in-case-number to case-number
    Move in-case-date   to case-date
    Move in-case-count  to case-count
    Move in-case-contents to case-contents
    Exit method.
END METHOD "SetInstanceData".

```

```

IDENTIFICATION DIVISION.
METHOD-ID. "GetInstanceData".
DATA DIVISION.
Linkage Section.
01 out-case.
    05 out-case-number    Pic 9(5).
    05 out-case-date      Pic X(8).
    05 out-case-count     Pic 99.
    05 out-case-contents.
        10 out-case-entry occurs 12 times.
            15 out-case-bottle Pic X(20).
PROCEDURE DIVISION Returning out-case.
    Move case-number to out-case-number
    Move case-date   to out-case-date
    Move case-count  to out-case-count
    Move case-contents to out-case-contents
    Exit method.
END METHOD "GetInstanceData".

```

```

IDENTIFICATION DIVISION.
METHOD-ID. "AddBott".
Same as Version 1
END METHOD "AddBott".

```

```
IDENTIFICATION DIVISION.  
METHOD-ID. "RemoveBott".  
Same as Version 1  
END METHOD "RemoveBott".
```

```
IDENTIFICATION DIVISION.  
METHOD-ID. "CalculateCost".  
Same as Version 1  
END METHOD "CalculateCost".
```

```
IDENTIFICATION DIVISION.  
METHOD-ID. "GetCaseNumber".  
Same as version 1  
END METHOD "GetCaseNumber".
```

```
IDENTIFICATION DIVISION.  
METHOD-ID. "DescribeCase".  
Same as Version 1  
END METHOD "DescribeCase".  
END CLASS Winecase.
```

---

## 12.2 Newcase Class Example

```
CLASS-ID. Newcase Inherits Winecase.  
ENVIRONMENT DIVISION.  
Configuration Section.  
Repository.  
    CLASS NewCase IS "NewCase"  
    CLASS Winecase IS "Winecase".  
DATA DIVISION.  
PROCEDURE DIVISION.  
*  
*  
*  
END CLASS Newcase.
```

---

## 12.3 Oldcase Class Example

CLASS-ID. OldCase Inherits Winecase.

ENVIRONMENT DIVISION.

Configuration Section.

Repository.

CLASS OldCase IS "OldCase"

CLASS Winecase IS "Winecase".

DATA DIVISION.

PROCEDURE DIVISION.

\*

\*

\*

```

IDENTIFICATION DIVISION.
METHOD-ID. "ReadCase".
ENVIRONMENT DIVISION.
Input-Output Section.
File-Control.
    SELECT the-case-file ASSIGN thecase
    Organization is Line Sequential.
DATA DIVISION.
File Section.
FD The-case-file External.
01 The-case-record    Pic X(255).
Working-Storage Section.
01 the-case.
    05 the-case-number    Pic 9(5).
    05 the-case-date      Pic X(8).
    05 the-case-count     Pic 99.
    05 the-case-contents.
        10 the-case-entry occurs 12 times.
            15 the-case-bottle    Pic X(20).
77 eof-flag    Pic 9.
88 eof    Value 0.
Linkage Section.
01 case-number    Pic 9(5).
PROCEDURE DIVISION Using Case-number.
    Open Input The-case-file
    Move 1 to eof-flag
    Perform until eof
        Read the-case-file into the-case
        At end
            Set eof to TRUE
        Not at end
            IF Case-number = the-case-number
                Invoke SELF "SetInstanceData" Using The-case
            END-IF
        End-Read
    End-Perform
    Close The-Case-file
    Exit Method.
END METHOD "ReadCase".

```

```

IDENTIFICATION DIVISION.
METHOD-ID. "Checkbott".
DATA DIVISION.
Working-Storage Section.
77 Random-setting Pic 9(8) Usage Comp.
77 sub Pic 99.
77 status-flag Pic 9.
   88 in-stock VALUE 0.
   88 out-stock VALUE 1.
01 the-case.
   05 the-case-number Pic 9(5).
   05 the-case-date Pic X(8).
   05 the-case-count Pic 99.
   05 the-case-contents.
       10 the-case-entry occurs 12 times.
       15 the-case-bottle Pic X(20).
Linkage Section.
01 out-contents.
   05 out-entry occurs 12 times.
   10 out-bottle Pic X(20).
01 out-count Pic 99.
PROCEDURE DIVISION Returning out-contents out-count.
Invoke SELF "GetInstanceData" Returning The-case
Move 0 to out-count
Perform varying sub from 1 by 1
    until sub > the-case-count
        Compute Random-setting = 0.5 + Function Random
        Compute Random-setting = Function Integer (Random-setting)
        IF Random-Setting = 1
            Set out-stock to TRUE
        ELSE
            Set in-stock to TRUE
        END-IF
        IF out-stock
            Add 1 to out-count
            Move The-case-bottle(sub) to out-bottle(out-count)
        END-IF
    End-Perform
Exit method.
END METHOD "Checkbott".
*
END CLASS Oldcase.

```



---

## 12.4 UserInterface Class Example

```
CLASS-ID. UserInterface Inherits SOMObject.
ENVIRONMENT DIVISION.
Configuration Section.
Repository.
    CLASS SOMObject IS "SOMObject"
    CLASS UserInterface IS "UserInt".
DATA DIVISION.
Working-Storage Section.
01  User-action      Pic X(10).
    88 User-add      Value "Addbott".
    88 User-delete   Value "Deletebott".
    88 User-end      Value "End".
01  User-Bottle      Pic X(20).
PROCEDURE DIVISION.
*
```

```
IDENTIFICATION DIVISION.
METHOD-ID. "ReadRequest".
DATA DIVISION.
Linkage Section.
01  Request          Pic X(6).
PROCEDURE DIVISION Returning Request.
    Display "Enter the request: new, status"
    Accept request from SYSIN
    Move Function Upper-case(request) to Request
    Exit Method.
END METHOD "ReadRequest".
```

```
IDENTIFICATION DIVISION.
METHOD-ID. "ReadInput1".
Same as "ReadInput" in Version 1
END METHOD "ReadInput1".
```

```
IDENTIFICATION DIVISION.
METHOD-ID. "ReadInput2".
DATA DIVISION.
Linkage Section.
01 Acct-num Pic 9(5).
PROCEDURE DIVISION Returning Acct-num.
    Display "Enter the account number."
    Accept Acct-num from SYSIN
    Exit Method.
END METHOD "ReadInput2".
```

```
IDENTIFICATION DIVISION.
METHOD-ID. "WriteMessage".
Same as version 1
END METHOD "WriteMessage".
```

```
IDENTIFICATION DIVISION.
METHOD-ID. "Writeoutput".
Same as version 1
END METHOD "Writeoutput".
```

```
IDENTIFICATION DIVISION.
METHOD-ID. "WriteStatus".
DATA DIVISION.
Working-Storage Section.
77 sub Pic 99 Value 99.
Linkage Section.
01 Out-table.
    05 Out-Entry occurs 12 times.
        10 Out-Bottle Pic X(20).
01 Out-count Pic 99.
PROCEDURE DIVISION Using Out-table Out-count.
    IF out-count > 0
        Perform varying sub from 1 by 1
            until sub > out-count
        Display "Out of stock " Out-Bottle(sub)
        End-Perform
    END-IF
    Exit Method.
END METHOD "WriteStatus".
END CLASS UserInterface.
```

## 12.5 Wine Client Program

```
Version 2
PROGRAM-ID. "Wine".
ENVIRONMENT DIVISION.
Configuration Section.
Repository.
    CLASS SOMObject    IS "SOMObject"
    CLASS NewCase      IS "NewCase"
    CLASS OldCase      IS "OldCase"
    CLASS UserInterface IS "UserInt".
DATA DIVISION.
Working-Storage Section.
77  univObj           Usage Object Reference.
77  userObj           Usage Object Reference UserInterface.
77  Case-number       Pic 9(5).
77  total-cost        Pic 9(7)V99.
77  out-count         Pic 9(2).
77  request           Pic X(6).
77  action            Pic X(10).
77  bottle            Pic X(20).
77  flag              Pic X.
01  Case-Contents.
    05 Case-Entry occurs 12 times.
        10 Case-Bottle Pic X(20).
PROCEDURE DIVISION.
    Invoke UserInterface "somNew" RETURNING userObj
    Invoke userobj "ReadRequest" Returning request.
    IF request = "STATUS"
        Perform CheckOldCase
    ELSE
        Perform CreateNewCase
    END-IF.
    Invoke userobj "somFree"
    STOP RUN.
```

CheckOldCase.

    Invoke OldCase "somNew" Returning univobj  
    Invoke userobj "ReadInput2" Returning Case-number  
    Invoke univobj "ReadCase" Using Case-number Returning flag  
    Invoke univobj "CheckBott" Returning Case-contents out-count  
    Invoke userobj "WriteStatus" Using Case-contents out-count  
    Invoke univObj "somFree".

CreateNewCase.

    Invoke NewCase "somNew" Returning univobj

    Invoke userobj "ReadInput1" Returning bottle action

    Perform until action = "End"

        IF action(1:3) = "Add"

            Invoke univObj "AddBott" Using bottle Returning flag

        ELSE

            Invoke univObj "DeleteBott" Using bottle Returning flag

        END-IF

    Invoke userObj "WriteMessage" Using flag

    Invoke userObj "ReadInput1" Returning bottle action

End-Perform

    Invoke univObj "CalculateCost" Returning total-cost

    Invoke univObj "GetCaseNumber" Returning case-number

    Invoke userObj "WriteOutput" Using total-cost case-number

    Invoke univObj "DescribeCase"

    Invoke univObj "somFree".

END PROGRAM "Wine".

---

## Chapter 13. Metaclasses

This chapter explains the concept of metaclasses in object-oriented COBOL programming.

---

### 13.1 Need for Metaclasses

Suppose you want to do:

- Operations on all objects of a class  
For example, count, calculate maximum, and list sequence
- Operations at creation  
For example, establish instance data and add a parameter

We could code in the client, but reuse would not be possible.

Up until now we have only considered activities with individual objects, for example updating data for a specific case of wine, that is, one case equals one object.

Suppose we want to do something with all the objects of a particular class? For example, we might want to count how many cases we have referenced or know the maximum cost of our cases (assuming that cost was something we stored). We might want to list all the dates that our cases were ordered.

It might be possible to write the routine in a client program but we would have to repeat the code in every client program.

Suppose when we create an object we want all its data to be loaded from a file. If it were the same data for each object in our class, there is no problem. We can override the `somDefaultInit` method in our class definition and put the code there.

But in practice that data would be dependent on some key value, such as a personnel number, an account number or a case reference number. We cannot pass that to `somDefaultInit` because it is not expecting a parameter.

Again, we could write a method which we explicitly invoke after creating the object (this is what we did in Example 2), but if we wanted to be sure this happened (for example, to write an audit record) must rely on the client program coder.

A guaranteed action might also be required on freeing up the object. This might or might not be possible by overriding `somUninit`.

---

### 13.2 Definition

The use of metaclasses answers the needs explained in the last section. The concept can be thought of as follows:

- Objects can represent anything, such as people, accounts, orders, cases of wine
- All objects have classes which define them

- Consider objects which represent classes
- Their defining classes are called METACLASSES

We create an object to represent all the objects of a particular type.

Just as we have an object to represent a case of wine, so we can have an object to represent that object. Just as a class holds the definitions for our case objects (the class Winecase), so we can have a class holding the definitions for our class objects (MetaWinecase).

### 13.3 Classes and Metaclasses Example

On the left side of Table 13 a class ClaretBottle inherits from a WineBottle class which inherits from a Container which inherits similarly from other classes until reaching SOMObject.

On the right side of Table 13 there is a metaclass for the ClaretBottle Class called MetaClaret. This inherits from a chain of classes culminating in one called SOMClass.

Method in WineBottle is available to ClaretBottle and method in SOMClass is available to MetaClaret and, in both cases, we can add methods or data to inheriting class.

Table 13. Classes and Metaclasses		
Classes		Metaclasses
ClaretBottle	#	
(inherits from)	#	MetaClaret
WineBottle	#	
(inherits from)	#	
Container	#	(inherits from)
...	#	
...	#	
...	#	SOMClass
SOMObject	#	

It might inherit directly from SOMClass, or there might be 100 classes inbetween. There is no need for there to be a metaclass for WineBottle or Container. Even if there were, there would be no need for those classes to be in the MetaClaret to SOMClass chain.

On the left-hand side any methods higher up the chain from ClaretBottle such as ListPrice or StateSupplier can be used on ClaretBottle and we can add methods or data to ClaretBottle which do not apply higher up the chain. For example, we might want for ClaretBottle to have a method EstimateValue and a new data value called Value.

The same is true for the right hand-side. To count instances,

1. Add data item COUNT to MetaClaret
2. Add method "CreateObj" to MetaClaret

Add one to COUNT  
Invoke "somNew" on itself

3. In the client, invoke "CreateObj" on MetaClaret  
(rather than "somNew" on ClaretBottle)

If we add a data item called COUNT and a method called CreateObj to MetaClaret, we can put code in that method which will increase the value of COUNT everytime the method is invoked, and then to create an instance of what Metaclaret defines. In other words, we create an object.

In a client program we normally would have invoked somNew on the class ClaretBottle. Instead, we invoke the method CreateObj on the metaclass MetaClaret. This has the same result as creating an instance of ClaretBottle, but on the way it also stores a count.

This is called a "constructor" method. There are "destructor" methods too which might, for example, reduce the value of COUNT by one.

---

## 13.4 Definition

The structure of a metaclass definition is like a class definition:

- ID Division.  
Class-ID. *Metaname* Inherits SOMClass.  
.....
- Environment Division.  
Repository.  
Class *Metaname* is "*Metaname*"  
Class SOMClass is "SOMClass".  
.....
- Data Division.  
Working-Storage Section.  
(No Local Storage, NO Linkage)  
(No Value clauses)
- Procedure Division.  
... (discussed later)
- End Class *Metaname*

Like any other class definition, a metaclass has the usual four divisions:

Identification  
Environment  
Data  
Procedure

The Procedure Division contains the definitions of the methods.

The ID division contains the Class-ID statement, with the next class in the inheritance chain between the metaclass itself and SOMClass. Typically this is SOMClass itself.

Whatever name is put where SOMClass, is must match the first name on the corresponding Class statement in the Repository section. Case is ignored. This is also true for the name of the metaclass itself, which must match the name specified on the End-Class statement after the procedure division.

Convention dictates that the actual names as defined in the SOM Repository be used as well.

As with class definitions, we have the Repository in the Environment Division linking our program references to classes in the actual SOM Interface Repository. The correct case must be used between the quote marks. The quote marks are single or double depending on the compiler option APOST/QUOTE setting.

There is only a Working-Storage Section in the Data Division. (The methods' Data Divisions may have all four). No Value may be specified on the definition statement. Whatever value is put in the data items remains until changed or until the very last instance of the class disappears. All data defined here is directly accessible to the methods of this metaclass, but not directly accessible to client programs or other classes and objects, which must access it using the methods.

#### Procedure Division (of Metaclass)

- Constructor Method - key statements

Data Division. (of method)

Linkage Section.

01 xyzObj USAGE OBJECT REFERENCE.

Procedure Division. (of method)

Procedure Division Using .. Returning xyzObj.

Invoke SELF "somNew" Returning xyzObj

The procedure division, as for a class, consists of method definitions. Everything that was described there applies here.

In the Linkage Section of the Data Division, a definition of an object is required. This can be a universal object, with an arbitrary name such as xyzObj.

In the Procedure Division, the division statement itself specifies that the object defined is returned. The executable statement Invoke must be coded specifying SELF (case is not significant) as the class and somNew as the method.

#### Class programs

- ID Division.

Class-ID. *Classname* INHERITS ... METACCLASS *Metaname*.

- Repository.

Class *Metaname* is "*Metaname*"

Class *Classname* is "*Classname*"



The ID Division specifies the metaclass as shown (METAClass and INHERITS do not need to be in uppercase. It is shown in capitals as a reminder that they are keywords. Similarly the METAClass metaname pair can come before the INHERITS ... pair.)

Client programs

- Invoking Constructor Methods

01 xyzObj USAGE OBJECT REFERENCE.

.....

Invoke *Classname* "CreateObj" Using ... Returning xyzObj

- Invoking Other Methods

01 pqrObj USAGE OBJECT REFERENCE METAClass *Metaname*.

.....

Invoke xyzObj "somGetClass" Returning pqrObj

Invoke pqrObj "OtherMethod" Using ... Returning ...

When invoking constructor methods, only a universal object definition in Working-Storage is required such as xyzObj. Then the method can be invoked from the procedure division against the classname. It is the name of the class, not the name of the metaclass that is specified. (We did not specify the Metaclass in the Repository section of this program, because we're not referencing it).

For invoking other methods in the metaclass we must point to them. We probably are already using an object derived from the class, perhaps using the constructor method. So we have a handle, such as xyzObj. We refer to the metaclass as pqrObj and define it as an object with the keyword METAClass (as ever, case is not important) and the name of the metaclass.

We then use the method somGetClass (which is defined in SOMObject and available in our object xyzObj using inheritance) against our object. The method returns the handle to the metaclass in pqrObj. We then specify pqrObj in our invoke statement, allowing us to access the method desired.



---

## Chapter 14. Example Three

This is the third and final version of our example. This example illustrates the use of metaclasses. Appendix C contains the full code listing.

In Version three the unchanged items are as follows:

- Winecase class
- Newcase class
- Userinterface class

---

### 14.1 MetaOldCase MetaClass

```
IDENTIFICATION DIVISION
CLASS-ID. MetaOldCase Inherits SOMClass.
ENVIRONMENT DIVISION.
Configuration Section.
Repository.
    CLASS MetaOldCase IS "MetaOldCase"
    CLASS OldCase IS "OldCase"
    CLASS SOMClass IS "SOMClass".
DATA DIVISION.
Working-Storage Section.
01 status-count    Pic 99.
PROCEDURE DIVISION.
*
*
*
IDENTIFICATION DIVISION.
METHOD-ID. somDefaultInit OVERRIDE.

PROCEDURE DIVISION.
    Move 0 to status-count.
    Exit Method.
END METHOD somDefaultInit.
```

```

IDENTIFICATION DIVISION.
METHOD-ID. CreateOldCase.
DATA DIVISION.
Linkage Section.
01 Case-number Pic 9(5).
01 anObj USAGE OBJECT REFERENCE.
PROCEDURE DIVISION Using Case-number Returning anObj.
    IF Case-number > 0
        Invoke SELF "somNew" Returning anObj
        Invoke anObj "ReadCase" Using Case-number
        Add 1 to status-count
    END-IF
    Exit method.
END METHOD CreateOldCase.

```

```

IDENTIFICATION DIVISION.
METHOD-ID. CountOldCase.
DATA DIVISION.
Linkage Section.
01 out-count Pic 9(2).
PROCEDURE DIVISION Returning out-count
    Move status-count to out-count.
    Exit method.
END METHOD CountOldCase.
END CLASS MetaOldCase.

```

---

## 14.2 OldCase Class

```

IDENTIFICATION DIVISION.
CLASS-ID. OldCase Inherits Winecase MetaClass MetaOldCase
ENVIRONMENT DIVISION.
Configuration Section.
Repository.
    CLASS MetaOldCase IS "MetaOldCase"
    CLASS OldCase IS "OldCase"
    CLASS Winecase IS "Winecase".
DATA DIVISION.
PROCEDURE DIVISION.
*
*
*

```

```

IDENTIFICATION DIVISION.
METHOD-ID. "ReadCase".
ENVIRONMENT DIVISION.
Input-Output Section.
File-Control.
    SELECT the-case-file ASSIGN thecase
    Organization is Line Sequential.
DATA DIVISION.
File Section.
FD The-case-file External.
01 The-case-record    Pic X(255).
Working-Storage Section.
01 the-case.
    05 the-case-number    Pic 9(5).
    05 the-case-date      Pic X(8).
    05 the-case-count     Pic 99.
    05 the-case-contents.
        10 the-case-entry occurs 12 times.
            15 the-case-bottle    Pic X(20).
77 eof-flag    Pic 9.
88 eof    Value 0.
Linkage Section.
01 case-number    Pic 9(5).
PROCEDURE DIVISION Using Case-number.
    Open Input The-case-file
    Move 1 to eof-flag
    Perform until eof
        Read the-case-file into the-case
        At end
            Set eof to TRUE
        Not at end
            IF Case-number = the-case-number
                Invoke SELF "SetInstanceData" Using The-case
            END-IF
        End-Read
    End-Perform
    Close The-Case-file
    Exit Method.
END METHOD "ReadCase".

```

```

IDENTIFICATION DIVISION.
METHOD-ID. "Checkbott".
DATA DIVISION.
Working-Storage Section.
77 Random-setting Pic 9(8) Usage Comp.
77 sub Pic 99.
77 status-flag Pic 9.
   88 in-stock VALUE 0.
   88 out-stock VALUE 1.
01 the-case.
   05 the-case-number Pic 9(5).
   05 the-case-date Pic X(8).
   05 the-case-count Pic 99.
   05 the-case-contents.
       10 the-case-entry occurs 12 times.
       15 the-case-bottle Pic X(20).
Linkage Section.
01 out-contents.
   05 out-entry occurs 12 times.
   10 out-bottle Pic X(20).
01 out-count Pic 99.
PROCEDURE DIVISION Returning out-contents out-count.
Invoke SELF "GetInstanceData" Returning The-case
Move 0 to out-count
Perform varying sub from 1 by 1
    until sub > the-case-count
        Compute Random-setting = 0.5 + Function Random
        Compute Random-setting = Function Integer (Random-setting)
        Display "Random number is " Random-setting
        IF Random-Setting = 1
            Set out-stock to TRUE
        ELSE
            Set in-stock to TRUE
        END-IF
        IF out-stock
            Add 1 to out-count
            Move The-case-bottle(sub) to out-bottle(out-count)
        END-IF
    End-Perform
Exit method.
END METHOD "Checkbott".
END CLASS Oldcase.

```

## 14.3 Wine Client Program

```
PROGRAM-ID. "Wine".
ENVIRONMENT DIVISION.
Configuration Section.
Repository.
    CLASS SOMObject    IS "SOMObject"
    CLASS NewCase      IS "NewCase"
    CLASS OldCase      IS "OldCase"
    CLASS UserInterface IS "UserInt".
DATA DIVISION.
Working-Storage Section.
77  univObj           Usage Object Reference.
77  metaObj           Usage Object Reference MetaClass OldCase.
77  userObj           Usage Object Reference UserInterface.
77  Case-number       Pic 9(5).
77  total-cost        Pic 9(7)V99.
77  out-count         Pic 9(2).
77  request           Pic X(6).
77  action            Pic X(10).
77  bottle            Pic X(20).
77  flag              Pic X.
01  Case-Contents.
    05 Case-Entry occurs 12 times.
        10 Case-Bottle Pic X(20).
PROCEDURE DIVISION.
Invoke UserInterface "somNew" RETURNING userObj
Invoke userobj "ReadRequest" Returning request.
IF request = "STATUS"
    Perform CheckOldCase
ELSE
    Perform CreateNewCase
END-IF.
Invoke userobj "somFree"
STOP RUN.
```

```

    CheckOldCase.
    Invoke userObj "ReadInput2" Returning Case-number
    Invoke OldCase "CreateOldCase" Using Case-number Returning univObj.
    Perform Until Case-number < 0
        Invoke univObj "CheckBott" Returning Case-Contents out-count
        Invoke userObj "WriteStatus" Using Case-Contents out-count
        Invoke userObj "ReadInput2" Returning Case-number
        Invoke OldCase "CreateOldCase" Using Case-number Returning univObj
    End-Perform

    Invoke univObj "somGetClass" Returning metaObj
    Invoke metaObj "CountOldCase" Returning out-count
    Invoke userObj "WriteMessage" Using out-count OMITTED
    Invoke metaObj "somFree".

```

```

CreateNewCase.
    Invoke NewCase "somNew" Returning univObj
    Invoke userObj "ReadInput1" Returning bottle action
    Perform until action = "End"
        IF action(1:3) = "Add"
            Invoke univObj "AddBott" Using bottle Returning flag
        ELSE
            Invoke univObj "RemoveBott" Using bottle Returning flag
        END-IF
        Invoke userObj "WriteMessage" Using flag
        Invoke userObj "ReadInput1" Returning bottle action
    End-Perform

    Invoke univObj "CalculateCost" Returning total-cost
    Invoke univObj "GetCaseNumber" Returning case-number
    Invoke userObj "WriteOutput" Using total-cost case-number
    Invoke univObj "DescribeCase"
    Invoke univObj "somFree".

    END PROGRAM "Wine".

```



---

## Part 3. Object-Oriented COBOL - An Example



---

## Chapter 15. The Wine Store Scenario

The approach adopted in this part of the book is to invent a business requirement and show how to turn that requirement into an actual running system.

Rather than duplicate material provided elsewhere, for example in the *"Getting Started"* manuals listed in the bibliography at the beginning, the content is designed to be complementary by providing an illustrative example. Commentary brings out more clearly the point of the illustration.

A number of iterations are presented in the development exercise. Initially, a "no-frills," "bare-bones" approach is implemented. This is followed by a redevelopment of the same facets of the application using SOM much more heavily. The third iteration builds on the second, and introduces inheritance. The fourth builds on the third iteration, and uses metaclasses.

After these four iterations, development may be considered complete. Fifth, sixth, and seventh iterations are presented for those wishing to delve deeper into the subject. The fifth iteration involves the creation of DLLs and LIBs for classes. The sixth involves the porting of the application to a workstation without the VisualAge for COBOL for OS/2 product. Finally, in the seventh iteration, classes are used to generate SOM IDL and are registered in the SOM Interface Repository.

In this example the business process largely generates the functional requirements for the system.

---

### 15.1 Background

Our customer is a distant uncle on our mother's side. He has turned his hobby into a business and operates a retailing establishment for wine. As his business has grown, his manual methods of order entry and inventory control have proven inadequate. At Mom's urging and in a less-than-lucid moment after a quality control check of his products, he has contracted us to build an application system for the operation of his wine retailing business.

Besides wine, our uncle has recently adopted computing as a hobby. He is enthralled with object technology and prefers that we adopt an object approach. He wants to maintain the system himself after we have finished so he feels that we should use a high-level, easy-to-use programming language.

---

### 15.2 Overview of the Business Process

Wine store customers order their selections by the bottle. An order consists of any number of bottles, each of which is of a certain type and cost. There cannot be more than one bottle of the same type and cost (our uncle believes in variety). In dealing with customers, the retailer may consult an old order to check the status of the order. Also, new orders may be created.

When checking an old order, the customer gives the case number and the order is checked to see if each bottle is in stock. A list is produced of any out-of-stock

items. This is done for as many orders as desired, and when terminated, the seller would like to know how many orders have been checked.

When creating a new order, the seller adds bottles, or deletes any that have been added by mistake. Once the order is complete, an order number is assigned, and its cost computed.

---

## Chapter 16. OO Analysis and Design Processes

In the analysis phase of developing an object-oriented application, the systems analyst may utilize any number of techniques and tools to formulate a design to meet the users' expectations. Generally, the underlying strategy of object-oriented analysis is to find out what steps the user performs to accomplish a task, what things he uses in performing these tasks, and then transform these procedures into objects and methods to be implemented in code. In the analysis phase, we model the problem by identifying objects that interact and behave according to the system requirements. In the design phase, we model the solution using the semantic classes and their interfaces, the application classes, and the utility classes we have identified.

---

### 16.1 Methodology

The methodology employed for this example draws on other established object-oriented analysis and design methodologies, such as use cases from Ivar Jacobson and CRC cards from Kent Beck. Other popular methodologies exist such as Booch, OMT by James Rumbaugh, and the Fusion method by the Hewlett-Packard Corp. Our methodology does not purport to champion one methodology over another. Such a treatise is beyond the scope of the task at hand. An underlying reason for our approach spares us the burden of learning the complex system of notation used by other methodologies. It also allows us to proceed, in a rather direct fashion, with creating a solution in code. The reader familiar with introductory object-oriented concepts should be comfortable with the steps employed.

#### 16.1.1 Analysis

To implement the sample application, we employ the following procedures in our analysis and design:

- Use cases.

In developing our use cases, we place ourselves in the user's position and draft textual descriptions of his procedures. Use cases can become a hierarchy, in which a high-level use case uses one or more lower-level use cases. Use cases consist of a set of actors or external agents, the usages of the system by the actors, and links between the actors and use cases. Obviously, some limit must be employed by the analyst, and a determination of when to move on to the next step must be made to avoid analysis paralysis. But, generally speaking, time spent in this phase is time well spent and pays dividends later in the development process.

- Object identification.

This step identifies the objects the actors used in the use cases.

After our use cases have been formulated, we can analyze them and extract the nouns that were used in their text. We examine these nouns with an eye towards making them objects. In this examination, some of them clearly sift out as objects, while others appear trivial and useless as objects. The analyst therefore faces a decision: is this item an object or is it an attribute of another object? It has been said that one man's object is another man's attribute.

- Method determination.

We parse the grammar of our use cases and identify the verbs used. These verbs then become candidates for the methods used to modify our objects and their attributes.

- CRC card formulation.

This step constructs CRC cards and their derivatives.

CRC is an acronym for "Class Responsibilities and Collaborations". For each class (or object), these cards list the responsibilities of the class and what other classes are used by the class in accomplishing these responsibilities. A derivative of the CRC card may be used which graphically represents the object, its attributes, and its methods in a doughnut form. These doughnuts have literally become a trademark of object-oriented analysis and design.

## 16.1.2 Design

Constructing object interaction diagrams is the first step in the design process. Then we move on to implementing the design in code.

Typically, we produce an object interaction diagram for each use case that we are implementing. These diagrams provide system designers an understanding of the message flow between objects and the sequence of events in the use case. A message to an object evokes a response from it. Object interaction diagrams are dynamic models of the application, while use cases are static.

At this point in the design effort, we have identified the objects, their attributes and methods, and their relationships to each other in the business process. We can then begin to implement our design in code. In doing so, we may generate new insights into the business process we are implementing and identify some shortcomings in our design.

As the analyst develops a deeper, understanding of the business process, omissions and flaws in his design may become evident. To achieve a high level of user acceptance and satisfaction for the system, it is important to incorporate these new insights into the system. The advantages of this iterative process are one of the hallmarks of object-oriented design and analysis.

---

## 16.2 Analyzing the Objects

In designing the application for the wine store, we use the methodology outlined above.

In the first iteration of our development effort, the resulting application prototypical in a sense. It is not "throw-away" work, but something upon which we can build and refine in subsequent iterations.

### 16.2.1 A Use Case of the WineStore Application

Our first analysis step is to observe the actions of sales personnel and interview them about the procedures they use in processing customer orders.

**Basic Use Case -- A New Order is Placed:** When the customer places a new order for bottles of wine, the salesperson notes the bottles the customer has selected on the order, assigns a unique number and the current date to the order, and then adds the bottles of wine the customer has selected. The order is a list of bottles selected. If the customer changes his mind about a bottle that has been selected, the salesperson removes it from the order. As the

salesperson adds the bottles to the order, he notes the cost of the bottle selected and the type of wine on the order. After all the selected bottles have been added to the order, the total cost of the selected bottles is computed, and the order is reviewed and filed. As a practical matter, we arbitrarily assume that the order may hold a maximum of 64 bottles. Given our clientele, this is certainly large enough to handle any orders.

**Objects Used in the WineStore:** After the use case has been drafted, we examine it to determine what objects are used in the business process.

We first list the nouns in the use case. We omit pronouns such as *its* and *he/she* since, in actual usage, these are merely tags or synonyms for previously used nouns. From the basic use case we have:

- Customer
- Order
- Bottle(s)
- Wine
- Salesperson
- Number (order number)
- Date (current date)
- Selections
- Mind
- Type
- Cost.

From this list, we consider “customer” and “mind.” A “customer” is external to our application and is identified by an “order” number, not by name or account. “Customer” is an actor in our system. Hence, “customer” is not a good candidate for object status. “Mind,” in this context, is an abstract state intrinsic to the cognitive decision-making processes of the customer; not an object in our system.

An “order” can be construed as a transaction in the business process. We can consider orders to be objects. It is an integral part of the application. An order is identified by a unique “number,” so that order number is an attribute of the order class. The current date is also assigned to the order, so we consider it an attribute of the order object.

“Salesperson” is an actor of our application, and the primary user of the system. We do not consider him an object. However, we can make our interface with him an object to isolate the business logic from the presentation logic in our application.

“Wine” is the substance in the “bottles” that we sell. It occurs in “types,” that is, a “bottle” contains a certain “type” of “wine.” Bottles are the items placed in a “container” ordered by the customer. Bottle is a good candidate for an object in our application; wine and type are interchangeable, and attributes of the bottle object. Selection is a synonym for bottle as used in the context of this application.

“Cost” is an attribute of the bottle object because it provides descriptive information about the object, and is somewhat intrinsic to a given type of wine. There is nothing mentioned about case prices or a multiple bottle pricing strategy.

We offer the following summary of our preliminary examination:

- Bottles are objects with attributes of type (of wine) and cost.
- Orders are objects having attributes of a unique identifying number, the current date, and a list of bottles.
- The user interface is an object, but all of the details concerning it may not arise until later in the analysis phase.

**Methods Used in the WineStore:** Having identified the objects to be used, we next determine the methods of the objects for further examination from the use cases and the objects themselves. We do this by listing the verbs used in the use cases, omitting various forms of the verb "to be." For clarity, we also include the direct object of the verb where possible.

From the use case, we find

- Places order
- Notes bottles
- Has selected
- Assigns (order) number
- Assigns date
- Adds bottles
- Changes mind
- Removes it (bottle)
- Notes type and cost
- Removes bottle
- Compute (cost)
- Reviews order
- Files order.

From this list, we identify actions that relate to our objects: bottle, order, and user interface. The remainder we discard.

"Places a new order" may be construed as instantiating, or creating, an order object, and beginning the process. Hence, we deduce that a constructor for the order object is needed.

"Assigns (order) number" indicates a need to set instance data for the order. Generally, if we set instance data for an object, we require a method to get the instance data in the process. Since orders are referenced by unique number, we need a method for getting the unique number.

"Assigns date" indicates a need for a method of setting the date attribute of the order method. We create a set method for this attribute as well.

"Notes bottles" is an action taken by the user of the system, and indicates a need to hold data about the selections in the order.

"Adds bottles" and "removes bottles" are other methods that we need for our order object.

"Compute (cost)" indicates a method must be written to calculate the cost of the order. To compute the cost of the order, the bottle objects must be capable of returning their cost. Hence, cost is an attribute of the bottle object.



"Reviews order" indicates a need for a method to describe the order after it has been completed. "Files order" means that we need a method for externalizing the order to some medium.

For the user interface object, methods are needed to accept input from the system user and to display system responses.

We can summarize classes with the following:

- WineOrder Class
  - Attributes:

<i>Table 14. WineOrder Class Attributes</i>	
<b>Attributes</b>	
Order number	
Order date	
Order contents (bottles)	

- Methods:

<i>Table 15. WineOrder Class Methods</i>	
Method	Purpose
Constructor	creates an instance of the object.
Destructor	destroys an instance of the object.
AddBottle	adds a bottle to the order.
Removebottle	deletes a bottle from the order.
CalculateCost	computes the cost of the bottles in the order.
DescribeOrder	describes the contents of the order.
GetOrderNumber	returns the order number of an order object.
GetOrderDate	returns the order date of an order object.
SetOrderNumber	sets the order number of an order object.
SetOrderDate	sets the order date of an order object.
XternOrder	writes the order to a file.

– CRC card:

-----	
Class: WineOrder	
-----	
Responsibilities:	Collaborators:
Add a bottle to the order	UIInterface
Remove a bottle from the order	UIInterface
Calculate the cost of order	WineBottle, UIInterface
Describe the contents of an order	WineBottle, UIInterface
Get the order number	UIInterface
Get the order date	UIInterface
Set the order number	UIInterface
Set the order date	UIInterface
Externalize the order	WineBottle

Figure 46. CRC Card for WineOrder Class

– WineBottle Class

<i>Table 16. WineBottle Class Attributes</i>
<b>Attributes</b>
Cost
Wine Type

– Methods:

<i>Table 17. WineBottle Class Methods</i>	
Method	Purpose
Constructor	creates an instance of the object.
Destructor	destroys an instance of the object.
GetCost	returns the cost of the bottle.
GetType	returns the type of wine in the bottle.
SetCost	sets the cost of the bottle.
SetType	sets the type of wine in the bottle.

- CRC card:

-----	
Class: WineBottle	
-----	
Responsibilities:	Collaborators:
Get the bottle cost	WineOrder
Get the wine type	WineOrder
Set the bottle cost	WineOrder
Set the wine type	WineOrder

Figure 47. CRC Card for WineBottle Class

- **UserInterface Class**

To isolate the business logic from the presentation logic, we can add an object that interfaces with the user of the application. This proves to be a useful strategy if we later decide to alter the user interface. The business logic can remain unchanged when making modifications to the user interface.

- Attributes:

<i>Table 18. UserInterface Class Attributes</i>
<b>Attributes</b>
Action
Bottle (selected)

- Methods:

<i>Table 19. UserInterface Class Methods</i>	
Method	Purpose
ReadAction	gets the input command from the system user.
ReadType	gets the type of wine from the system user.
ReadCost	gets the cost of the bottle from the system user.
WriteMessage	displays a system status message to the user.
WriteOutput	displays the cost of the order and the order number to the user.
WriteBottle	displays attributes of a bottle collected in the order to the user.

– CRC card:

----- Class:  UserInterface -----	
Responsibilities:	Collaborators:
Accept a request from the system user:	
--Add a bottle to order	WineOrder, WineBottle
--Remove a bottle from order	WineOrder, WineBottle
Respond to the system user:	
--Display a status message	WineOrder
--Display the order cost and the order number	WineOrder, WineBottle
--Display the order contents	WineOrder, WineBottle

Figure 48. CRC Card for UserInterface Class

---

## Chapter 17. Object-Oriented COBOL Implementation

---

### 17.1 Code Creation Process

For the first phase of our wine store business example, we implement the WineOrder class with its AddBott, DeleteBott, CalculateCost, GetOrderNumber, GetOrderDate, and DescribeOrder methods. We also implement the UserInterface class with its ReadInput, WriteMessage, and Writeoutput methods. The WineBottle class with its GetCost and GetType methods are created. Finally, we code a client program that uses a UserInterface object to discover what bottles need to be added to an Order.

---

### 17.2 COBOL Code Creation Structure

We have identified the attributes (instance data) and methods needed for the first iteration of our classes. Winebottle is the class referred to by the Order object, while the user is connected to the system using the UserInterface class. The whole is tied together by the Wine client. This is the sequence with which we will work on the development.

Coding consists of the following methods for a given class:

- Constructor
- Set method for each attribute of the instance data
- Get method for each attribute of the instance data
- Destructor
- Copy constructor

Since we are using SOM, we program constructors, destructors, and copy constructors. Since we use more of the functionality of SOM, some situations require us to override the basic constructors and destructors of SOM.

Those familiar with object-oriented design patterns (or Smalltalk) should recognize our system as an implementation of a model-view-controller (MVC) pattern. The model comprises the WineBottle and WineOrder classes, the view is the UserInterface class, and the controller is the system object.

The model classes could interface with the view, thereby saving an iteration of returning to the controller to interface with the view. However, this is contrary to the MVC paradigm, in which the controller controls both the model and the view. It may be necessary for model classes to communicate with each other, but they should never cross the model boundary and communicate with the interface. The corollary is also true: the interface should not invoke model classes directly, but should work through the controller. Strictly adhering to this pattern provides encapsulation of the view class; the underlying rationale is that different views may be added depending on requirements, and that the model and controller classes will not require changes.

**Coding Process:** Alternative approaches to the coding part of the development process might have been adopted, for example using the Visual Builder. However, in order to provide a minimum of distraction in this illustrative case, we decided to use a command line interaction with the familiar cycle as follows:

1. Code the class programs and the client program using a standard text or program editor.
2. Compile them all using the command:

```
COB2 client.CBL cls1.CBL cls2.CBL cls3.CBL cls4.CBL
```

where "cls1", "cls2" and so on are the class programs.

The client program must be specified first but thereafter the order is not significant. The COB2 command requires no specific options. although "-g" is required if the debugger is expected to be used.

The names before the extensions of the class source files are entirely arbitrary. They need have no relationship to the names of the classes as referenced either at the COBOL level or at the SOM level. However, the name of the client COBOL source determines (unless overridden) the name of the .EXE file eventually produced, no matter what the PROGRAM-ID statement declares in the source of the program.

We could also create a compile.cmd file that contains the lengthy command needed for compilation.

3. Correct any compilation errors.
4. Recompile using the COB2 command, specifying .CBL for programs previously in error and .OBJ for the others.

If we coded a compile.cmd file, we can, of course, use it, recompiling all modules once again. The COB2 command compiles any .CBL files and then links all the files, making the first program specified the invocable program, with a file type of .EXE.

**Test the Client Program:** Even if the client program compiled clean, it must be tested.

Use either "Display" statements or a debugger to discover the causes of any aberrant behavior. To use the debugger, the "-g" parameter must be specified on the COB2 command or "process test" must be specified as the first line of the COBOL source program.

---

## 17.3 Code Commentary

The following sections contain extracts of the important features of the client and class programs, with descriptions of how the code works.

### 17.3.1 Wine Client

The Wine client program interacts with an object from the user interface class and one from the order class to create an order.

The Identification Division appears as follows:

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. "Wine".
```

The Environment Division appears as follows:

```

ENVIRONMENT DIVISION.
Configuration Section.
Repository.
    CLASS SOMObject      IS "SOMObject"
    CLASS TheOrder       IS "WineOrder"
    CLASS UserInterface IS "UserInt".

```

The new statement, Repository, introduces the list of classes the program will reference. Specifically, this program links the terms used subsequently to refer to the classes, for example TheOrder and UserInterface, with the names as SOM knows them, for example "WineOrder" and "UserInt".

SOM is case sensitive while COBOL is not. This means that in the CLASS statements, case is important only for letters between the quote marks. For example, the "W" and the "O" in "WineOrder" must be in uppercase while the other letters must be in lower case. Elsewhere, case can be used in whatever way the developer is most comfortable. Thus "TheOrder" could be "theorder", "THEORDER", or even "ThEoRdEr" if required.

The last CLASS statement is concluded with a full-stop or period. All others must not end this way.

The Data Division has no Linkage Section or Local-Storage Section, thus leaving only a Working-Storage Section. The first part appears as follows:

```

DATA DIVISION.
Working-Storage Section.
01  orderObj      Usage Object Reference TheOrder.
01  userObj       Usage Object Reference UserInterface.

```

In the executable part of the code, in the Procedure Division, objects are referred to using these definitions. For example, objects from the class "TheOrder" are referenced using the term "orderObj". This is called a handle. Similarly "userObj" is a handle for objects of the UserInterface class.

Important definitions in the remainder of the Working-Storage are as follows:

```

01  Item-Count      Pic 9(4).
01  Max-items       Pic 9(4) Value 64.
01  Order-number    Pic 9(5).
01  Order-date      Pic X(8) Value "00/00/00".

```

<b>Item-count</b>	This is the number of bottles in the order.
<b>Max-items</b>	This is used to limit the number of bottles in the order to 64.
<b>Order-number</b>	This is an identification for the order, created with a random number generator.
<b>Order-date</b>	The date of order creation is captured in this variable, in YY/MM/DD form.

All the remaining definitions are only work items used briefly in various operations.

```

77  Item-type       Pic X(20).
77  Item-cost       Pic 999V99.
77  total-cost      Pic 9(7)V99.
77  action          Pic X(10).
77  Today           Pic X(21).
77  flag            Pic X.

```

```

01  WS-items.
   05 WS-count      Pic 9(4).
   05 WS-item       Occurs 1 to 64 times
                     Depending on WS-count
                     Indexed by WS-Index.
       10  WS-type   Pic X(20).
       10  WS-cost   Pic 999V99.

```

Finally, the Procedure Division works as follows:

#### PROCEDURE DIVISION.

1. Create an object for the user interface.

```
Invoke UserInterface "somNew" RETURNING userObj
```

The COBOL verb used in interacting with an object is Invoke. It has the following parameters:

- The name of the object
- The object method being invoked
- Data being passed to the object (with a "Using" clause)
- Data being returned (with a "Returning" clause).

This example is a special case, since the object is not one belonging to the program but one more belonging to the system. It is the object representing the Class UserInterface and the only reason the client program Wine interacts with it is to create an object userObj.

The section on metaclasses contains a fuller and more rigorous description of these special objects. The important point is that to create an instance of an object it is necessary to use the Invoke verb, specifying the somNew method and the class of interest, and supplying an object reference handle so that the object returned can be subsequently used.

2. Create an object for the order.

```
Invoke TheOrder      "somNew" RETURNING orderObj
```

This is another example of the same type of special case.

3. Initialize with a number and the date.

```

Move Function Current-date to Today
Move Today(3:2) to Order-date(1:2)
Move Today(5:2) to Order-date(4:2)
Move Today(7:2) to Order-date(7:2)
Compute Order-Number
      = Function Integer(10000 * Function Random )
Invoke orderObj "InitOrder" Using Order-Number Order-Date

```

This is a more typical use of Invoke. The object representing the order is told to execute the method InitOrder and to use the data held in variables Order-Number and Order-Date. There is no data to be returned and so there is no Returning clause.

The program defining the class WineOrder from which the object orderObj was generated is described later.

Remember that case is ignored in COBOL (with the one exception noted earlier) and in particular the method name inside the quotes, InitOrder. It can have any combination of upper and lower case letters. The combination chosen does not need to match those used in the method's definition in the class program, nor does it need to be consistent within a client program if used more than once.



4. Item-count contains the number of bottles in the order. Set it to zero.

Move Zero to Item-Count

5. The user is asked whether to add a bottle to the order, to delete one, or to end the application.

Invoke userObj "ReadAction" Returning action  
Perform until action = "End"  
or Item-Count >= Max-Items

6. When adding, the type and cost of the wine are requested.

IF action(1:3) = "Add"  
Add 1 to Item-Count  
Invoke userObj "Readtype" Returning Item-type  
Invoke userObj "ReadCost" Returning Item-cost

7. The Wineorder object maintains a list of bottles already ordered. The order object is told to add a bottle object to that list with the type and cost supplied. A confirmation message is supplied using the user interface object.

Invoke orderObj "AddBott" Using Item-type Item-cost  
Returning flag  
Invoke userObj "WriteMessage" Using flag

8. When deleting the type and cost of the wine are requested to identify the correct bottle to delete.

ELSE  
Invoke userObj "Readtype" Returning Item-type  
Invoke userObj "Readcost" Returning Item-cost  
Invoke orderObj "DeleteBott"  
Using Item-type Item-cost Returning Flag  
Invoke userObj "WriteMessage" Using flag  
Subtract 1 from Item-count  
END-IF

9. The question is then repeated.

Invoke userObj "ReadAction" Returning action  
End-Perform

10. When user input is complete, the cost of the order is calculated and printed.

Invoke orderObj "CalculateCost" Returning total-cost  
Invoke orderObj "GetOrderNumber" Returning Order-number  
Invoke orderObj "GetOrderDate" Returning Order-date  
Invoke userObj "WriteOutput"  
Using total-cost Order-number Order-Date

11. All the individual bottle details are printed.

Invoke orderObj "DescribeOrder" Returning WS-items  
Invoke userObj "Writebottle" Using WS-items

12. The Objects are then destroyed.

Invoke orderObj "ScrapOrder"  
Invoke orderObj "somFree"  
Invoke userObj "somFree"

Currently COBOL does not clean up objects at the end of the run-unit so it is good practice to issue the somFree method against all objects created.

While the opposite method of somNew is invoked against the class object, in this case Bottle, somFree is invoked against the object itself.

```
STOP RUN.  
END PROGRAM "Wine".
```

### 17.3.2 WineOrder Class

The program defining the class "Wine-Order" has the following three data attributes:

```
Order-number  
Order-date  
Bottles.
```

It has the following eight methods:

```
DescribeOrder  
Addbott  
Deletebott  
Calculatecost  
GetOrderDate  
GetOrderNumber  
ScrapOrder  
InitOrder.
```

Many of these access those attributes.

The Identification Division appears as follows:

```
IDENTIFICATION DIVISION.  
CLASS-ID. "WineOrder" Inherits SOMObject.
```

There are other parameters possible on the CLASS-ID statement, but the only mandatory one is the immediate ancestor of the class being defined, in this case SOMObject.

The Environment Division appears as follows:

```
ENVIRONMENT DIVISION.  
Configuration Section.  
Repository.  
    CLASS SOMObject IS "SOMObject"  
    CLASS Bottle    IS "WineBottle"  
    CLASS WineOrder IS "WineOrder".
```

As with the client program Wine, it is necessary to list all classes referenced directly. The class we are defining, WineOrder, does not absolutely have to be mentioned here. It must be all upper-case, no matter how it has been typed in the CLASS-ID statement earlier. This means that every other program that refers to the class must use upper-case letters between the quote marks in their CLASS statements.

The first name in such a CLASS statement (that is, one referring to the class being defined) should match the class-id statement.

One full-stop or period must be at the end of the Repository paragraph.

The Data Division appears as follows:

```

DATA DIVISION.
Working-Storage Section.
01 Order-Number      Pic X(5).
01 Order-date        Pic X(8).
01 bottles.
    05 Bottle-count   Pic 9(4).
    05 Bottle-item    Occurs 1 to 64 times
                        Depending on Bottle-count
                        Indexed by Item-Count.
    10 bottleObj      Usage Object Reference Bottle.

```

Each order requires a five digit number to identify it. The date the order was taken is recorded in YY/MM/DD form.

The Object Reference illustrated here, to objects of the class Bottle, demonstrates that object references can be members of a structure, just like other data types. The handle bottleObj is part of the structure Bottles because up to sixty-four bottles are allowed in the order. In this initial implementation there will be a bottleObj object for each bottle in the order.

The Procedure Division contains only method definitions, as follows.

#### PROCEDURE DIVISION.

##### 1. DescribeOrder

```

IDENTIFICATION DIVISION.
METHOD-ID. "DescribeOrder".
DATA DIVISION.
Local-Storage Section.
01 WS-Type            Pic x(20).
01 WS-Cost            Pic 999V99.
Linkage Section.
01 LS-Items.
    05 LS-Item-Count   Pic 9(4).
    05 LS-Item         Occurs 1 to 64 times
                        Depending on LS-Item-Count
                        Indexed by LS-Index.
    10 LS-Type         Pic x(20).
    10 LS-Cost         Pic 999V99.

```

This Data Division introduces the new concept of Local Storage. Variables are re-allocated every time the method is invoked, unlike Working Storage where the values remain in their last-used state.

```

PROCEDURE DIVISION Returning LS-items.
    Move Bottle-count to LS-Item-Count
    Set LS-Index to 1
    Perform varying Item-Count from 1 by 1
        until (Item-Count > Bottle-count)
        Invoke bottleObj(Item-Count) "GetCost"
            Returning WS-Cost
        Move WS-Cost to LS-Cost (LS-Index)
        Invoke bottleObj(Item-Count) "GetType"
            Returning WS-Type
        Move WS-Type to LS-Type (LS-Index)
        Set LS-Index up by 1
    End-Perform
    Exit method.
END METHOD "DescribeOrder".

```

This method produces a table of bottles showing their types and their costs. This takes advantage of the fact that the Bottle objects have methods to return those attributes.

## 2. AddBott

```
IDENTIFICATION DIVISION.
METHOD-ID. "AddBott".
DATA DIVISION.
Working-Storage Section.
01  Found-Flag      Pic X.
   88  found        VALUE "0".
   88  not-found    VALUE "1".
Linkage Section.
01  LS-Type        Pic X(20).
01  LS-Cost        Pic 999V99.
01  LS-flag        Pic X.
```

These variables in Working-Storage can be defined in Local Storage, as it happens.

```
PROCEDURE DIVISION USING LS-Type LS-Cost
    Returning LS-flag.
    Move "1" to LS-flag
    Found-flag
    Perform varying Item-Count from 1 by 1
        until (Item-Count > 64) or (found)
        If BottleObj(Item-Count) = NULL
            Invoke Bottle "somNew"
                Returning BottleObj(Item-Count)
            Invoke bottleObj(Item-Count) "SetType"
                Using LS-Type
            Invoke bottleObj(Item-Count) "SetCost"
                Using LS-Cost
            Add 1 to Bottle-Count
            Move "0" to LS-flag
            Found-Flag
        END-IF
    End-Perform.
    Exit method.
END METHOD "AddBott".
```

This method exploits the fact that an object can be NULL. The first time this method is invoked, no Bottle objects will have been created. Thus the first handle tested for being NULL proves positive and the object is created. On subsequent occasions, the method tests several handles before finding one that is NULL.

After the bottle object is created, values are assigned to its two attributes using methods written specially for the purpose. The method also has attributes to reveal the values and the Get-Attribute and Set-Attribute methods are very common.

## 3. DeleteBott

```
IDENTIFICATION DIVISION.
METHOD-ID. "DeleteBott".
DATA DIVISION.
Working-Storage Section.
01  Found-Flag      Pic X.
   88  found        VALUE "0".
   88  not-found    VALUE "1".
```

```

Local-Storage Section.
77  Bott-Type      Pic X(20).
77  Bott-Cost      Pic 999V99.
Linkage Section.
01  LS-Type        Pic X(20).
01  LS-Cost        Pic 999V99.
01  Delete-flag    Pic x.
PROCEDURE DIVISION USING LS-Type LS-Cost
    Returning Delete-flag.
    Move "1" to Found-Flag
    Delete-flag
    Perform varying Item-Count from 1 by 1
        until (Item-Count > bottle-count) or (found)
        Invoke BottleObj(Item-Count) "GetType"
            Returning Bott-type
        If LS-type = Bott-type
            Invoke BottleObj(Item-Count) "GetCost"
                Returning Bott-cost
            IF LS-Cost = Bott-cost
                Set BottleObj(Item-Count) to BottleObj(Bottle-count)
                Set BottleObj(bottle-count) to NULL
                Subtract 1 from Bottle-Count
                Move "0" to Delete-flag
                Found-Flag
            END-IF
        END-IF
    End-Perform.
    Exit method.
END METHOD "DeleteBott".

```

The algorithm used involves working through the list of bottle objects, checking to see if the type and cost attributes match those required. When they do, the verb SET is used to make the object in the list since it is no longer required to be the same as the last object in the list. Then, SET is used again to make that last object NULL. That is, it no longer represents a bottle as far as the application is concerned. It is still available for use, however, as demonstrated in the AddBott method.

#### 4. CalculateCost

```

IDENTIFICATION DIVISION.
METHOD-ID. "CalculateCost".
DATA DIVISION.
Local-Storage Section.
77  cost  Pic 999V99.
Linkage Section.
01  LS-Total-cost  Pic 9(7)V99.
PROCEDURE DIVISION Returning LS-Total-cost.
    Move 0 to LS-Total-cost
    Perform varying Item-Count from 1 by 1
        until Item-Count > Bottle-count
        Invoke bottleObj(Item-Count) "GetCost" Returning Cost
        ADD Cost    to LS-Total-cost
    End-Perform.
    Exit method.
END METHOD "CalculateCost".

```

#### 5. GetOrderDate

```

IDENTIFICATION DIVISION.
METHOD-ID. "GetOrderDate".
DATA DIVISION.
Linkage Section.
01  LS-Order-Date          Pic X(8).
PROCEDURE DIVISION Returning LS-Order-date.
    Move Order-date to LS-Order-date.
    Exit method.
END METHOD "GetOrderDate".

```

#### 6. GetOrderNumber

```

IDENTIFICATION DIVISION.
METHOD-ID. "GetOrderNumber".
DATA DIVISION.
Linkage Section.
01  LS-Order-Number        Pic X(5).
PROCEDURE DIVISION Returning LS-Order-Number.
    Move Order-number to LS-Order-number.
    Exit method.
END METHOD "GetOrderNumber".

```

Get-attribute and "Set-attribute" methods were mentioned earlier, in conjunction with the Bottle class. This last example shows two examples of Get-attribute methods used in this class.

#### 7. ScrapOrder

```

IDENTIFICATION DIVISION.
METHOD-ID. "scrapOrder".
DATA DIVISION.
Local-Storage Section.
PROCEDURE DIVISION.
    Subtract 1 from bottle-Count
    Perform varying Item-Count from bottle-count by -1
        until (Item-Count = 0)
        Invoke bottleObj(Item-Count) "somFree"
    End-Perform
    Exit method.
END METHOD "scrapOrder".

```

#### 8. "InitOrder"

```

IDENTIFICATION DIVISION.
METHOD-ID. "InitOrder".
DATA DIVISION.
Linkage Section.
01  LS-Order-Number        Pic X(5).
01  LS-Order-Date          Pic X(8).
PROCEDURE DIVISION Using LS-Order-Number LS-Order-Date.
    Move LS-Order-Number to Order-number
    Move LS-Order-Date to Order-date
    Exit Method.
END METHOD "InitOrder".
*
*
END CLASS "WineOrder".

```

An alternative to this method is to have two Set-attribute methods. However these particular attributes are only set once, when the object is created, so it is clearer to group the actions using Init method.

### 17.3.3 UserInterface Class

The UserInterface object handles the interaction between the client program, Wine, and the rest of the system.

```
IDENTIFICATION DIVISION.
CLASS-ID. "UserInterface" Inherits SOMObject.
ENVIRONMENT DIVISION.
Configuration Section.
Repository.
  CLASS SOMObject IS "SOMObject"
  CLASS UserInterface IS "UserInterface".
```

The start of the definition is similar to those already discussed.

```
DATA DIVISION.
Working-Storage Section.
01  User-action      Pic X(10).
    88  User-add      Value "Addbott".
    88  User-delete   Value "Deletebott".
    88  User-end      Value "End".
```

The only item in Working-Storage for the whole object is referenced both from outside the class and by different methods within the class.

```
PROCEDURE DIVISION.
```

There are six methods defined:

- ReadAction
- ReadType
- Readcost
- WriteMessage
- WriteOutput
- WriteBottle

#### 1. ReadAction

```
IDENTIFICATION DIVISION.
METHOD-ID. "ReadAction".
DATA DIVISION.
Linkage Section.
01  Action      Pic X(10).
PROCEDURE DIVISION Returning Action.
  Display "Enter the action : add, delete, end"
  Accept action from SYSIN
  Move Function Upper-case(action) to Action
  Evaluate action
    When "ADD"
      Set User-add to TRUE
    When "DELETE"
      Set User-delete to TRUE
    When "END"
      Set User-end to TRUE
  End-evaluate
  Move User-action to action
  Exit Method.
END METHOD "ReadAction".
```

## 2. ReadType

```
IDENTIFICATION DIVISION.
METHOD-ID. "ReadType".
DATA DIVISION.
Working-Storage Section.
01  WS-type      Pic X(80).
Linkage Section.
01  LS-Type      Pic X(20).
PROCEDURE DIVISION Returning LS-Type.
    Display "Enter the item"
    Accept WS-Type from SYSIN
    Move WS-Type(1:20) to LS-Type
    Exit Method.
END METHOD "ReadType".
```

## 3. ReadCost

```
IDENTIFICATION DIVISION.
METHOD-ID. "ReadCost".
DATA DIVISION.
Working-Storage Section.
01  WS-Cost      Pic X(6).
Linkage Section.
01  LS-Cost      Pic 999V99.
PROCEDURE DIVISION Returning LS-Cost.
    Display "Please enter the cost: "
    Accept WS-Cost from SYSIN
    Compute LS-Cost = Function Numval-c (WS-Cost)
    Exit Method.
END METHOD "ReadCost".
```

## 4. WriteMessage

```
IDENTIFICATION DIVISION.
METHOD-ID. "WriteMessage".
DATA DIVISION.
Linkage Section.
01  LS-Flag      Pic X.
PROCEDURE DIVISION Using LS-Flag.
    IF LS-flag = "0"
        Display user-action " successfully completed"
    ELSE
        Display user-action " unsuccessfully completed"
    END-IF.
    Exit Method.
END METHOD "WriteMessage".
```

## 5. WriteOutput

```
IDENTIFICATION DIVISION.
METHOD-ID. "Writeoutput".
DATA DIVISION.
Working-Storage Section.
77  Formatted-cost Pic $,ZZZ,ZZ9.99.
Linkage Section.
01  Total-cost    Pic 9(7)V99.
01  Order-number  Pic 9(5).
01  Order-date    Pic X(8).
PROCEDURE DIVISION Using Total-cost Order-number Order-date.
    Move total-cost to Formatted-cost
    Display "Your order costs " Formatted-cost
    Display "Your order number is " Order-number
```



```

        Display "Your order date is "   Order-date
        Exit Method.
    END METHOD "Writeoutput".

```

#### 6. WriteBottle

```

    IDENTIFICATION DIVISION.
    METHOD-ID. "Writebottle".
    DATA DIVISION.
    Working-Storage Section.
    01  WS-Formatted-Cost   Pic ZZ9.99.
    Linkage Section.
    01  LS-items.
        05 LS-count        Pic 9(4).
        05 LS-item         Occurs 1 to 64 times
                           Depending on LS-count
                           Indexed by LS-Index.
        10 LS-type         Pic X(20).
        10 LS-cost         Pic 999V99.

    PROCEDURE DIVISION Using LS-items.
        Perform varying LS-Index from 1 by 1
            until LS-Index > LS-Count
                Move LS-Cost(LS-Index) to WS-Formatted-Cost
                Display LS-Type(LS-Index) " at " WS-Formatted-Cost
        End-Perform
        Exit Method.
    END METHOD "Writebottle".

    END CLASS "UserInterface".

```

### 17.3.4 Bottle Class

The Bottle class is used to represent bottles in an order.

```

    IDENTIFICATION DIVISION.
    CLASS-ID. "Bottle" Inherits SOMObject.

    ENVIRONMENT DIVISION.
    Configuration Section.
    Repository.
        CLASS SOMObject IS "SOMObject"
        CLASS Bottle IS "Bottle".

```

The Identification and Environment Divisions are similar to those shown previously.

```

    DATA DIVISION.
    Working-Storage Section.
    01  Wine-Type          Pic X(20).
    01  Wine-cost          Pic 999V99.

```

Two important attributes for the application are the type of wine and its cost.

```

    PROCEDURE DIVISION.

```

There are five methods

- GetType
- SetType
- GetCost
- SetCost

- InitBott

1. GetType

```
IDENTIFICATION DIVISION.  
METHOD-ID. "GetType".  
ENVIRONMENT DIVISION.  
DATA DIVISION.  
Linkage Section.  
01 LS-Type          Pic X(20).  
PROCEDURE DIVISION Returning LS-Type.  
    Move Wine-Type to LS-Type  
    Exit method.  
END METHOD "GetType".
```

This is a classic example of a method supplying the value of an attribute, as are the next three methods.

2. SetType

```
IDENTIFICATION DIVISION.  
METHOD-ID. "SetType".  
ENVIRONMENT DIVISION.  
DATA DIVISION.  
Linkage Section.  
01 LS-Type          Pic X(20).  
PROCEDURE DIVISION Using LS-Type.  
    Move LS-Type to Wine-Type  
    Exit method.  
END METHOD "SetType".
```

3. GetCost

```
IDENTIFICATION DIVISION.  
METHOD-ID. "GetCost".  
ENVIRONMENT DIVISION.  
DATA DIVISION.  
Linkage Section.  
01 LS-Cost          Pic 999V99.  
PROCEDURE DIVISION Returning LS-Cost.  
    Move Wine-Cost to LS-Cost  
    Exit method.  
END METHOD "GetCost".
```

4. SetCost

```
IDENTIFICATION DIVISION.  
METHOD-ID. "SetCost".  
ENVIRONMENT DIVISION.  
DATA DIVISION.  
Linkage Section.  
01 LS-Cost          Pic 999V99.  
PROCEDURE DIVISION Using LS-Cost.  
    Move LS-Cost to Wine-Cost  
    Exit method.  
END METHOD "SetCost".
```

5. InitBott

```

IDENTIFICATION DIVISION.
METHOD-ID. "InitBott" .
DATA DIVISION.
Linkage Section.
01 LS-Type          Pic X(20).
01 LS-Cost          Pic 999V99.
PROCEDURE DIVISION Using LS-Type LS-Cost.
    Move LS-Type to Wine-Type
    Move LS-Cost to Wine-Cost
    Exit Method.
END METHOD "InitBott".
*
END CLASS "Bottle".

```

The last method, InitBott, has the same function as SetType and SetCost combined.

This section illustrated a working application modelling a subset of the business process functionality. More work still needs to be done, so we extend the application to come closer to the user's requirements.



---

## Chapter 18. The Second Iteration

Our uncle, fired with the enthusiasm of a first-time encounter with computers, has been reading technical magazines. He is now showing clear symptoms of "a little learning is a dangerous thing."

He has come across the concept of IBM's System Object Model (SOM), which has impressed him. As a result he has stated that his system must exploit SOM facilities. We explain that SOM offers great benefit, such as the ability to mix objects written in different languages and the support to enhance individual components of systems without requiring wholesale recompilations. We tell him that using IBM's VisualAge for COBOL for OS/2 means he is already able to take advantage of the benefits.

Unfortunately he has been overdoing the quality control checks of his own product line again and is quite adamant: "Either that system uses some of these SOM classes or vendor discounts become invalid."

Driven, as ever, by customer requirements we examine our code for an opportunity to work in SOM. We focus on the bottle objects currently held in a structure by our order object. Perhaps we can use a SOM Collection Class to provide the same function.

We introduce some extra error checking and convert some of our code to upper-case.

---

### 18.1 Code Commentary

This section contains extracts of the important features of the client and class programs, with descriptions of how the code works.

#### 18.1.1 A New Class

For this iteration, we add a new class: FileRW. To isolate the file I/O logic, the FileRW class handles the necessary file reads and writes. Separating this logic from other classes may be beneficial if we decide to adopt a database, change the file structure, or use a persistence framework, later in the application evolution. Other application classes remain unchanged when the file class is modified. Table 20 illustrates the second iteration methods.

- Attributes:
  - None
- Methods:

Table 20. Wine Store Scenario Methods for Second Iteration	
Method	Purpose
XternOrder	Writes the order to a flat file.
Constructor	Creates an instance of the file object.
Destructor	Destroys an instance of the file object.

- CRC card:

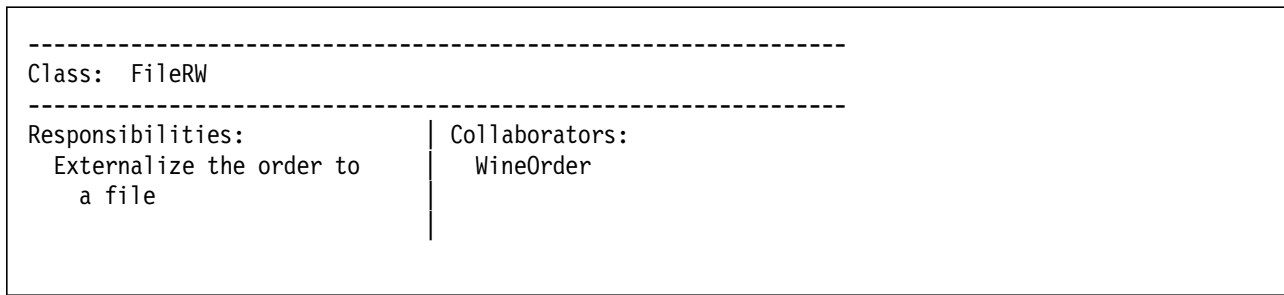


Figure 49. CRC Card for Wine Store Scenario Second Iteration

## 18.1.2 Object Interaction Diagram

To see how our objects interact with each other, we create a diagram which shows the flow from one object to another for the business process as shown in Figure 50.

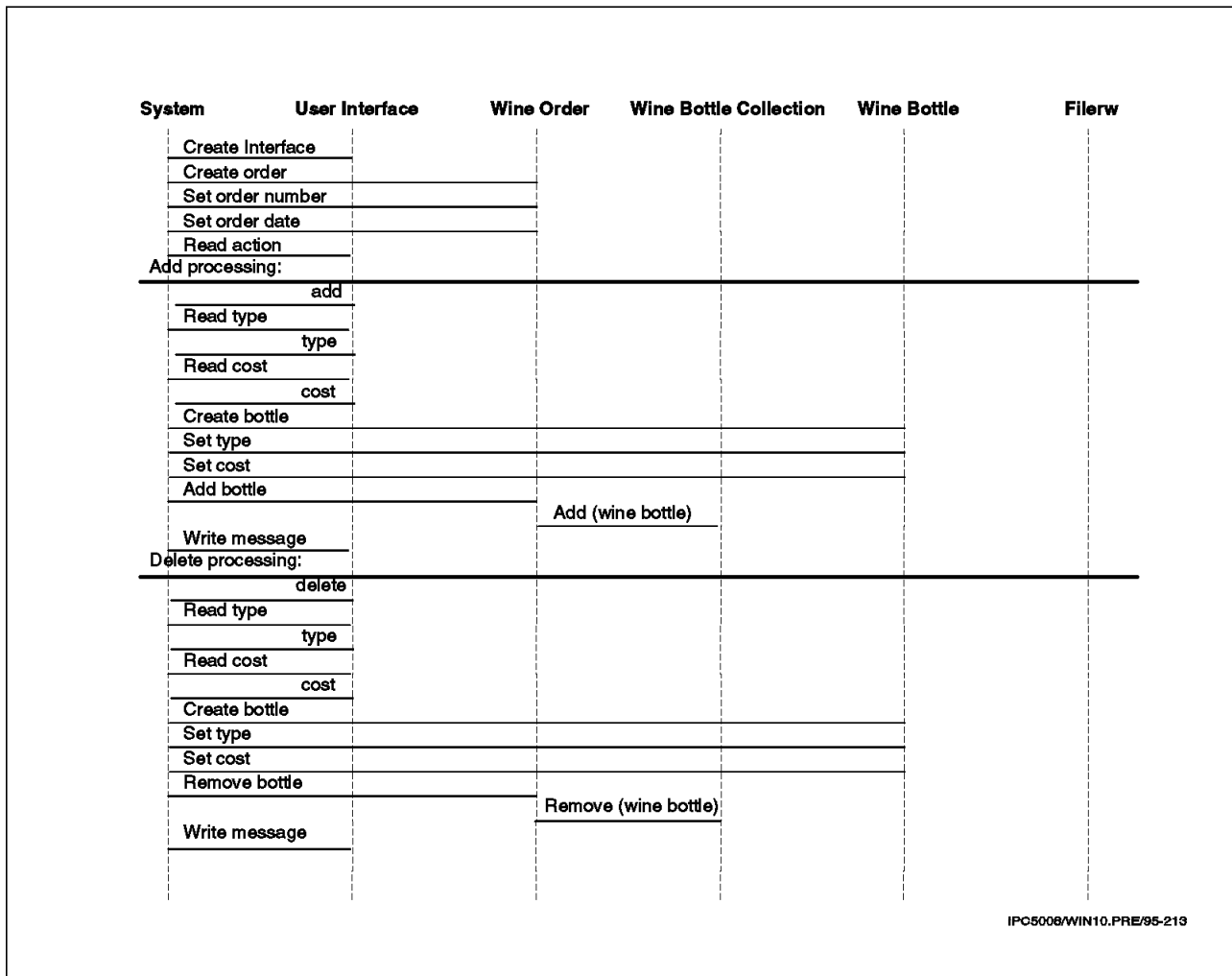


Figure 50. Object Interaction Diagram – Second Iteration

### 18.1.3 Wine Client

Phase2 of the Wine program uses SOM facilities to manipulate the bottle objects. It also adds test and reporting functions.

When using a SOM collection class, an object is returned when we iterate through the collection. This is important to remember because we must invoke get and set methods on the returned object to access any of its attributes.

```
process pgmname(mixed)
```

The process statement specifies compiler options. We set the pgmname option to mixed. The default of upper would normally be adequate, but if SOM programs are to be called, for example, somGetGlobalEnvironment as is used later, the option must be set to mixed.

```
IDENTIFICATION DIVISION.  
PROGRAM-ID.      "Wine".  
ENVIRONMENT DIVISION.  
CONFIGURATION SECTION.  
REPOSITORY.  
    CLASS SOMObject          IS "SOMObject"  
    CLASS WineOrder          IS "WineOrder"  
    CLASS UserInterface      IS "UserInterface".  
  
    CLASS Bottle             IS "WineBottle"  
    CLASS FileRW             IS "FileRW"
```

There are two new references to Classes in the client program Wine. One is to a new Class FileRW and the other to the WineBottle class. This latter class existed before, but its references were confined to the "Order" object.

```
DATA DIVISION.  
WORKING-STORAGE SECTION.  
01 orderObj          USAGE OBJECT REFERENCE WineOrder.  
01 userObj           USAGE OBJECT REFERENCE UserInterface.  
01 bottleObj         USAGE OBJECT REFERENCE Bottle.  
01 fileObj           USAGE OBJECT REFERENCE FileRW.
```

Only the new data definitions are listed below:

```
01 WS-PARMS.  
    05 ITEM-COUNT          PIC S9(8)          COMP.  
    05 WS-FLAG             PIC X.  
        88 SUCCESSFUL          VALUE "0".  
        88 FAILURE            VALUE "1".  
  
PROCEDURE DIVISION.  
    INVOKE UserInterface "somNew"    RETURNING userObj.  
    MOVE FUNCTION CURRENT-DATE TO ORDER-DATE.  
    COMPUTE WS-RANDOM-VAL = FUNCTION RANDOM.  
    COMPUTE ORDER-NUMBER = WS-RANDOM-VAL * 10000.  
  
    MOVE ZERO TO ITEM-COUNT.  
    INVOKE WineOrder "somNew"        RETURNING orderObj.
```

The order is initialized this time by its two explicit Set methods as opposed to its InitOrder method. This is required because SOM does not allow us to pass attributes to during object initialization using somNew. Hence, we create the object with somNew, and then use set methods for its attributes.

```
INVOKE orderObj "SetOrderNumber" USING ORDER-NUMBER.  
INVOKE orderObj "SetOrderDate"   USING ORDER-DATE.
```

```

        INVOKE userObj "ReadAction" RETURNING ACTION.
        PERFORM UNTIL ACTION = "END"
            OR ITEM-COUNT = MAX-ITEMS

```

```

        EVALUATE ACTION (1:3)
            WHEN "ADD"
                INVOKE userObj "ReadType" RETURNING ITEM-TYPE
                INVOKE userObj "ReadCost" RETURNING ITEM-COST

```

The client program itself generates and initializes a Bottle object, before adding it to the order. The RETURNING clause on the AddBottle invocation passes a structure, not an elementary data item. This technique is used when multiple data items must be returned, since RETURNING returns only a single item. We are expecting back a flag indicating the success or failure of the operation and the item count.

```

                INVOKE Bottle "somNew" RETURNING bottleObj
                INVOKE bottleObj "SetType" USING ITEM-TYPE
                INVOKE bottleObj "SetCost" USING ITEM-COST
                INVOKE orderObj "AddBottle" USING bottleObj
                    RETURNING WS-PARMS

```

If the add failed, the object just created is destroyed. The need for this is obvious as it is not in the collection, and we can't do anything with it. We should clean up the objects we create.

```

                IF WS-FLAG = "1"
                    INVOKE bottleObj "somFree"
                END-IF
                INVOKE userObj "WriteMessage" USING WS-FLAG

```

For delete processing, we retrieve the bottle type and cost, and then create a bottle object with these attributes. The newly created bottle object is then passed to the RemoveBottle method. Upon return, the bottle object is destroyed, and the user informed of the success or failure of the operation. The need to create an object like the one we are removing will become evident when we review the bottle and order classes.

```

            WHEN "DEL"
                INVOKE userObj "ReadType" RETURNING ITEM-TYPE
                INVOKE userObj "ReadCost" RETURNING ITEM-COST
                INVOKE Bottle "somNew" RETURNING bottleObj
                INVOKE bottleObj "SetType" USING ITEM-TYPE
                INVOKE bottleObj "SetCost" USING ITEM-COST
                INVOKE orderObj "RemoveBottle" USING bottleObj
                    RETURNING
                        WS-PARMS
                INVOKE bottleObj "somFree"
                INVOKE userObj "WriteMessage" USING WS-FLAG
            WHEN OTHER
                CONTINUE
        END-EVALUATE
        INVOKE userObj "ReadAction" RETURNING ACTION
    END-PERFORM.

```

A check is made of the number of bottles in the order. We don't want to perform any operations on an empty collection.

```

        IF ITEM-COUNT = 0
            THEN GOBACK.

```



```

INVOKE orderObj "CalculateCost"      RETURNING TOTAL-COST.
INVOKE orderObj "GetOrderNumber"    RETURNING ORDER-NUMBER.
INVOKE orderObj "GetOrderDate"      RETURNING ORDER-DATE.
INVOKE userObj  "WriteOutput"        USING TOTAL-COST
                                     ORDER-NUMBER
                                     ORDER-DATE.
INVOKE orderObj "DescribeOrder"      RETURNING WS-ITEMS.
INVOKE userObj  "WriteBottle"        USING WS-ITEMS.

```

The order is written to a file using a new object from the Class FileRW, and its method XternOrder.

```

INVOKE FileRW   "somNew"              RETURNING fileObj.
INVOKE fileObj  "XternOrder"          USING   orderObj.
INVOKE fileObj  "somFree".
INVOKE userObj  "somFree".
INVOKE orderObj "somFree".

GOBACK.
END PROGRAM "Wine".

```

### 18.1.4 WineOrder Class

The WineOrder class uses SOM facilities to maintain a list of Bottle objects.

```

process pgmname(mixed) test
  IDENTIFICATION DIVISION.
  CLASS-ID.    "WineOrder" INHERITS SOMObject.

```

The following methods were used in Version One:

- GetOrderNumber
- GetOrderDate
- SetOrderNumber
- SetOrderDate
- DescribeOrder
- CalculateCost
- AddBottle
- RemoveBottle

The following methods override the default methods supplied with SOM and inherited from SOMObject.

**somDefaultInit**      Initializes a WineOrder object.

**somFree**              Frees bottles, collection, and order.

```

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
  CLASS SOMObject          IS "SOMObject"
  CLASS WineBottle        IS "WineBottle".

  CLASS SOMCollection      IS "somf_TSet"
  CLASS SOMIterator        IS "somf_TSetIterator"

```

These two classes are supplied with SOM. A set is a type of collection. While we could order our collection, that is not necessary in this instance, so a set is used. The objects are maintained in the set in no particular order. somf\_TSet is one of the SOM collection classes.

```

DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-EV                      USAGE POINTER.
01 WINEORDER-OBJECT.
   05 WINEORDER-NUMBER        PIC X(5).
   05 WINEORDER-DATE          PIC X(8).
   05 WINEORDER-LIST          USAGE OBJECT REFERENCE SOMCollection.
01 WINEORDER-ITERATOR          USAGE OBJECT REFERENCE SOMIterator.

```

The handles WineOrder-List and WineOrder-Iterator are set up in the method called as part of this object's creation. The method somDefaultInit is described below.

```
PROCEDURE DIVISION.
```

#### 1. somDefaultInit

```

IDENTIFICATION DIVISION.
METHOD-ID. "somDefaultInit"    OVERRIDE.

```

We must specify that the method overrides a previously defined method, as inherited from SOMObject.

```

DATA DIVISION.
PROCEDURE DIVISION.
  CALL "somGetGlobalEnvironment" RETURNING WS-EV.

```

This call provides a pointer to the global environment structure of SOM. Exception data passes between most SOM methods and their invokers. Thus it is possible for the invoker to check for the success of its call. This technique is not used in these programs for the sake of clarity. A global environment pointer is required for a number of the SOM methods used in this example.

```

  INVOKE SOMCollection "somNew"
                        RETURNING WINEORDER-LIST.
  INVOKE WINEORDER-LIST "somfCreateIterator"
                        USING      BY VALUE WS-EV
                        RETURNING WINEORDER-ITERATOR.

```

This method is called to create an order object. In turn, two more objects are created whose handles are WineOrder-List and WineOrder-Iterator. WineOrder-List inherits from the SOM Collection class and holds the collection of bottles in the order. To pass through the collection, an iterator object is used. Use of the iterator is shown in some of the following methods.

```

  EXIT METHOD.
END METHOD "somDefaultInit".

```

#### 2. somFree

```

IDENTIFICATION DIVISION.
METHOD-ID. "somFree"          OVERRIDE.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 CollectedBottle          USAGE OBJECT REFERENCE WineBottle.
01 ITEM-COUNT                 PIC S9(8)    COMP.
PROCEDURE DIVISION.
  INVOKE WINEORDER-LIST "somfDeleteAll"
                        USING      BY VALUE WS-EV.

```

The WineOrder-List object knows about all the bottle objects created in the process of processing the order, and the somfDeleteAll method removes them.

```

        INVOKE WINEORDER-ITERATOR "somFree".
        INVOKE WINEORDER-LIST "somFree".
        INVOKE SUPER "somFree".

```

The two objects created in the somDefaultInit method, WineOrder-Iterator and WineOrder-Flag, are also removed, before the Order object removes itself.

SUPER invokes the method as defined in this class's superclass SOMObject. It is used for a method not overridden in the current class's definition. In this example, if SELF had been used instead of SUPER, a recursive invocation is executed, and a loop results. This is legal, but not required here.

```

        EXIT METHOD.
    END METHOD "somFree".

```

### 3. GetOrderNumber

This is essentially the same as in Version One.

### 4. GetOrderDate

This is essentially the same as in Version One.

### 5. SetOrderNumber

```

IDENTIFICATION DIVISION.
METHOD-ID. "SetOrderNumber".
DATA DIVISION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
01  LS-ORDERNUMBER          PIC X(5).
PROCEDURE DIVISION          USING    LS-ORDERNUMBER.
    MOVE LS-ORDERNUMBER TO WINEORDER-NUMBER.
    EXIT METHOD.
END METHOD "SetOrderNumber".

```

This is a conventional method to set an attribute, such as OrderNumber. The next method performs the same task for OrderDate. In version One only one method, InitOrder, performed these two functions.

### 6. SetOrderDate

```

IDENTIFICATION DIVISION.
METHOD-ID. "SetOrderDate".
DATA DIVISION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
01  LS-ORDERDATE           PIC X(8).
PROCEDURE DIVISION          USING    LS-ORDERDATE.
    MOVE LS-ORDERDATE TO WINEORDER-DATE.
    EXIT METHOD.
END METHOD "SetOrderDate".

```

### 7. DescribeOrder

```

IDENTIFICATION DIVISION.
METHOD-ID. "DescribeOrder".
DATA DIVISION.
WORKING-STORAGE SECTION.
01  CollectedBottle       USAGE OBJECT REFERENCE WineBottle.
01  WS-TYPE                PIC X(20).
01  WS-COST                PIC 999V99.
01  ITEM-COUNT             PIC S9(8)  COMP.

```

```

LINKAGE SECTION.
01 LS-ITEMS.
   05 LS-ITEM-COUNT          PIC S9(4).
   05 LS-ITEM                 OCCURS 1 TO 64 TIMES
                              DEPENDING ON LS-ITEM-COUNT
                              INDEXED BY LS-INDEX.
      10 LS-TYPE              PIC X(20).
      10 LS-COST              PIC 999V99.
PROCEDURE DIVISION
   RETURNING LS-ITEMS.
   INVOKE WINEORDER-LIST "somfCount"
   USING BY VALUE WS-EV
   RETURNING ITEM-COUNT.

```

WineOrder-List is a handle defined in the working storage for the whole class referring to the set of objects, Bottle objects in this case. somfCount is a method to supply the number of objects in that set.

```

   MOVE ITEM-COUNT TO LS-ITEM-COUNT.
   IF ITEM-COUNT > 0
      THEN SET 1 TO LS-INDEX
      INVOKE WINEORDER-ITERATOR "somfFirst"
      USING BY VALUE WS-EV
      RETURNING CollectedBottle

```

The somfFirst method invoked against the WineOrder-Iterator object, also defined in the class's working-storage, returns the first object in the set.

```

      PERFORM GET-TYPE-N-COST
   END-IF.

```

The GET-TYPE-N-COST routine obtains the Type and the Cost for the bottle retrieved from the set. The next portion of code does the same for the remaining bottles. It uses the somfNext method against the WineOrder-Iterator handle to obtain the next Bottle object in the set. We use somfFirst to retrieve the first object, and then use somfNext for the remainder. We cannot start with somfNext.

```

   SUBTRACT 1 FROM ITEM-COUNT.
   IF ITEM-COUNT > 0
      THEN PERFORM ITEM-COUNT TIMES
      ADD 1 TO LS-INDEX
      INVOKE WINEORDER-ITERATOR "somfNext"
      USING BY VALUE WS-EV
      RETURNING CollectedBottle
      PERFORM GET-TYPE-N-COST
   END-PERFORM
   END-IF.
   EXIT METHOD.
GET-TYPE-N-COST.
   INVOKE CollectedBottle "GetType"
   RETURNING WS-TYPE
   MOVE WS-TYPE TO LS-TYPE (LS-INDEX)
   INVOKE CollectedBottle "GetCost"
   RETURNING WS-COST
   MOVE WS-COST TO LS-COST (LS-INDEX).
END METHOD "DescribeOrder".

```

## 8. CalculateCost

```

IDENTIFICATION DIVISION.
METHOD-ID. "CalculateCost".
DATA DIVISION.
WORKING-STORAGE SECTION.
01 CollectedBottle    USAGE OBJECT REFERENCE WineBottle.
01 ITEM-COUNT          PIC S9(8)    COMP.
01 WS-COST             PIC 999V99.
LINKAGE SECTION.
01 LS-COST             PIC 9(7)V99.
PROCEDURE DIVISION
    RETURNING    LS-COST.
    MOVE ZERO TO LS-COST.

```

LS-Cost accumulates the total-cost.

```

    INVOKE WINEORDER-LIST "somfCount"
    USING      BY VALUE WS-EV
    RETURNING ITEM-COUNT.

```

As before, somfCount against the handle WineOrder-List returns the number of Bottle objects in the list.

```

    IF ITEM-COUNT > 0
        INVOKE WINEORDER-ITERATOR "somfFirst"
        USING      BY VALUE WS-EV
        RETURNING CollectedBottle

```

As before, somfFirst against the handle WineOrder-Iterator returns the first Bottle object in the list.

```

        PERFORM GET-COST
    END-IF.
    SUBTRACT 1 FROM ITEM-COUNT.
    IF ITEM-COUNT > 0
        THEN PERFORM ITEM-COUNT TIMES
            INVOKE WINEORDER-ITERATOR "somfNext"
            USING      BY VALUE WS-EV
            RETURNING CollectedBottle

```

As before, somfNext against the handle WineOrder-Iterator returns the next Bottle object in the list.

```

        PERFORM GET-COST
    END-PERFORM
    END-IF.
    EXIT METHOD.
    GET-COST.
        INVOKE CollectedBottle "GetCost"
        RETURNING WS-COST
        ADD WS-COST TO LS-COST.
    END METHOD "CalculateCost".

```

#### 9. AddBottle

```

IDENTIFICATION DIVISION.
METHOD-ID. "AddBottle".
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-BEFORE-COUNT    PIC S9(8)    COMP.
01 WS-AFTER-COUNT     PIC S9(8)    COMP.
01 CollectedBottle    USAGE OBJECT REFERENCE WineBottle.
01 theEqualFlag       PIC X.
01 ITEM-FOUND-FLAG    PIC X.
01 ITEM-COUNT         PIC S9(8)    COMP.
01 LOOP-COUNT         PIC S9(8)    COMP.

```

```

LINKAGE SECTION.
01 LS-BOTTLE          USAGE OBJECT REFERENCE WineBottle.
01 LS-PARMS.
    05 LS-ITEM-COUNT    PIC S9(4).
    05 LS-FLAG          PIC X.
PROCEDURE DIVISION
    USING      LS-BOTTLE
    RETURNING LS-PARMS.

    MOVE LOW-VALUE      TO ITEM-FOUND-FLAG.
    INVOKE WINEORDER-LIST "somfCount"
    USING      BY VALUE WS-EV
    RETURNING WS-BEFORE-COUNT.

```

As before, this provides the number of objects in the set.

```

    MOVE WS-BEFORE-COUNT TO ITEM-COUNT.
    IF ITEM-COUNT NOT = 0
        THEN INVOKE WINEORDER-ITERATOR "somfFirst"
            USING      BY VALUE WS-EV
            RETURNING CollectedBottle
        PERFORM CHECK-EQUAL
    END-IF
END-IF.
SUBTRACT 1 FROM ITEM-COUNT.
IF ITEM-COUNT > 0
    THEN PERFORM VARYING LOOP-COUNT
        FROM 1 BY 1
        UNTIL LOOP-COUNT > ITEM-COUNT
        OR ITEM-FOUND-FLAG = HIGH-VALUE
        INVOKE WINEORDER-ITERATOR "somfNext"
            USING      BY VALUE WS-EV
            RETURNING CollectedBottle
        PERFORM CHECK-EQUAL
    END-IF
END-PERFORM
END-IF.

```

This code checks whether the bottle to be added is already in the list. A loop counter is required to end the loop when we have found an equal object, or, after checking all the objects in the set. Hence, we don't want to perform item-count times. (We assume for illustrative purposes that no more than one bottle of the same type and cost is required in any one order. Our uncle believes in variety).

```

    IF ITEM-FOUND-FLAG = LOW-VALUE
        THEN INVOKE WINEORDER-LIST "somfAdd"
            USING BY VALUE WS-EV
            BY VALUE LS-BOTTLE.

```

The method somfAdd adds the supplied object to the list maintained by the object WineOrder-List.

```

    INVOKE WINEORDER-LIST "somfCount"
    USING      BY VALUE WS-EV
    RETURNING WS-AFTER-COUNT.
    MOVE WS-AFTER-COUNT TO LS-ITEM-COUNT.

```

Next, we check that the adding of the object to the list was successful by comparing the count before and after.

```

IF WS-BEFORE-COUNT = WS-AFTER-COUNT
    THEN MOVE "1" TO LS-FLAG
ELSE
    MOVE "0" TO LS-FLAG
END-IF.
EXIT METHOD.
PERFORM CHECK-EQUAL.
    INVOKE CollectedBottle "somfIsEqual"
        USING BY VALUE WS-EV
              BY VALUE LS-BOTTLE
              RETURNING theEqualFlag
    IF theEqualFlag = HIGH-VALUE
        THEN MOVE HIGH-VALUE TO ITEM-FOUND-FLAG.

```

This is the SOM way of testing whether two objects are the same. somfIsEqual is an overridden method in the bottle class. In version one we used the COBOL test of "If object1 = object2 ...".

```
END METHOD "AddBottle".
```

#### 10. RemoveBottle

```

IDENTIFICATION DIVISION.
METHOD-ID. "RemoveBottle".
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-BEFORE-COUNT          PIC S9(8)    COMP.
01 WS-AFTER-COUNT           PIC S9(8)    COMP.
01 CollectedBottle         USAGE OBJECT REFERENCE WineBottle.
01 theEqualFlag             PIC X.
01 ITEM-COUNT               PIC S9(8)    COMP.
01 LOOP-COUNT              PIC S9(8)    COMP.
LINKAGE SECTION.
01 LS-BOTTLE                USAGE OBJECT REFERENCE WineBottle.
01 LS-PARMS.
    05 LS-ITEM-COUNT         PIC S9(4).
    05 LS-FLAG              PIC X.
PROCEDURE DIVISION
    USING LS-BOTTLE
    RETURNING LS-PARMS.

    INVOKE WINEORDER-LIST "somfCount"
        USING BY VALUE WS-EV
        RETURNING WS-BEFORE-COUNT.
    MOVE WS-BEFORE-COUNT TO ITEM-COUNT.
    IF ITEM-COUNT NOT = 0
        THEN INVOKE WINEORDER-ITERATOR "somfFirst"
            USING BY VALUE WS-EV
            RETURNING CollectedBottle
        PERFORM CHECK-EQUAL-N-REMOVE
    END-IF.
END-IF.

```

This logic is now repeated for each object in the set until the required object is found and removed.

```

SUBTRACT 1 FROM ITEM-COUNT.
IF ITEM-COUNT > 0
    THEN PERFORM VARYING LOOP-COUNT
        FROM 1 BY 1
        UNTIL LOOP-COUNT > ITEM-COUNT
            OR theEqualFlag = HIGH-VALUE
        INVOKE WINEORDER-ITERATOR "somfNext"
            USING BY VALUE WS-EV

```

```

                                RETURNING CollectedBottle
                                PERFORM CHECK-EQUAL-N-REMOVE
                                END-IF
                                END-PERFORM
                                END-IF.

```

The same check of counts before and after is performed in the method AddBottle is now made:

```

.
.
.
                                CHECK-EQUAL-N-REMOVE.
                                INVOKE CollectedBottle "somfIsEqual"
                                USING    BY VALUE WS-EV
                                BY VALUE LS-BOTTLE
                                RETURNING theEqualFlag
                                IF theEqualFlag = HIGH-VALUE
                                THEN INVOKE WINEORDER-LIST
                                "somfRemove"
                                USING BY VALUE WS-EV
                                BY VALUE CollectedBottle

```

The method somfRemove invoked against WineOrder-List removes the bottle from the set.

```

                                INVOKE CollectedBottle "somFree".
                                EXIT METHOD.
                                END METHOD "RemoveBottle".

```

11. The class definition ends.

```

                                END CLASS "WineOrder".

```

### 18.1.5 UserInterface Class

The User Interface class only has a few minor differences from the first version.

```

process pgmname(mixed) test
    IDENTIFICATION DIVISION.
    CLASS-ID. "UserInterface"    INHERITS SOMObject.
    ENVIRONMENT DIVISION.
    CONFIGURATION SECTION.
    REPOSITORY.
        CLASS SOMObject        IS "SOMObject".
    DATA DIVISION.
    WORKING-STORAGE SECTION.
    01 USER-ACTION                PIC X(10).
        88 UA-ADD                VALUE "Add".
        88 UA-DELETE            VALUE "Delete".
        88 UA-END                VALUE "End".
    PROCEDURE DIVISION.
    IDENTIFICATION DIVISION.

```

1. ReadAction

```

        METHOD-ID. "ReadAction".
        DATA DIVISION.
        WORKING-STORAGE SECTION.
        01 WS-EDIT-FLAG        PIC X.
        LINKAGE SECTION.
        01 LS-ACTION            PIC X(10).
        PROCEDURE DIVISION
            RETURNING    LS-ACTION.

```



```

MOVE LOW-VALUE TO WS-EDIT-FLAG.
PERFORM UNTIL WS-EDIT-FLAG NOT = LOW-VALUE
    DISPLAY "Enter the action desired: add, delete, end: "
    ACCEPT USER-ACTION FROM SYSIN
    MOVE FUNCTION UPPER-CASE (USER-ACTION) TO USER-ACTION
    MOVE USER-ACTION TO LS-ACTION
    EVALUATE USER-ACTION (1:3)
        WHEN "ADD"
            MOVE HIGH-VALUE TO WS-EDIT-FLAG
        WHEN "DEL"
            MOVE HIGH-VALUE TO WS-EDIT-FLAG
        WHEN "END"
            MOVE HIGH-VALUE TO WS-EDIT-FLAG
        WHEN OTHER
            DISPLAY "Requested action was " USER-ACTION
            DISPLAY "Try again, fumblefingers!!!"
    END-EVALUATE
END-PERFORM.
EXIT METHOD.
END METHOD "ReadAction".

```

## 2. ReadType

This is unchanged from version one.

## 3. ReadCost

This is unchanged from version one.

## 4. WriteMessage

This is unchanged from version one.

## 5. WriteOutPut

This is unchanged from version one.

## 6. WriteBottle

This is unchanged from version one.

END CLASS "UserInterface".

## 18.1.6 WineBottle Class

The Bottle object provides the same methods as previously, but adds one method, `somflsEqual`, overridden from the supplied default.

```

process pgmname(mixed) test
    IDENTIFICATION DIVISION.
    CLASS-ID. "WineBottle" INHERITS somf-MCollectible.

```

Instead of the `SOMObject`, the `WineBottle` object inherits from `somf-Mcollectible` so that it can be manipulated by `SOM Collection` class methods.

```

    ENVIRONMENT DIVISION.
    CONFIGURATION SECTION.
    REPOSITORY.
        CLASS WineBottle IS "WineBottle"

```

```

        CLASS somf-MCollectible    IS "somf_MCollectible".
DATA DIVISION.
WORKING-STORAGE SECTION.
01  WINEBOTTLE-OBJECT.
    05  WINE-TYPE                PIC X(20).
    05  WINE-COST                PIC 999V99.

```

As before, the Bottle object has two data attributes, type and cost.

The method `somfIsEqual` tests whether two bottle objects are equivalent, as determined by whether their types and costs are the same.

```

PROCEDURE DIVISION.
IDENTIFICATION DIVISION.
METHOD-ID. "somfIsEqual"          OVERRIDE.
DATA DIVISION.
LOCAL-STORAGE SECTION.
01  ITEMTYPE                    PIC X(20).
01  ITEM COST                  PIC 999V99.
LINKAGE SECTION.
01  LS-EV                      USAGE POINTER.
01  theBottle                  Usage Object Reference WineBottle.
01  theEqualFlag               PIC X.

```

The success of the comparison is reflected in the `theEqualFlag`.

```

PROCEDURE DIVISION          USING BY VALUE LS-EV
                                BY VALUE theBottle
                                RETURNING theEqualFlag.
        INVOKE theBottle     "GetType"  RETURNING ITEMTYPE.
        INVOKE theBottle     "GetCost"   RETURNING ITEM COST.
        IF (WINE-TYPE = ITEMTYPE) AND
           (WINE-COST = ITEM COST)
            THEN MOVE HIGH-VALUE TO theEqualFlag
        ELSE
            MOVE LOW-VALUE TO theEqualFlag.
        EXIT METHOD.
END METHOD "somfIsEqual".

```

`theBottle` is the object compared to this instance, so its type and cost are obtained and compared.

All four other methods, `Get Cost`, `GetType`, `SetCost`, and `SetType` are unchanged from Version One.

```

END CLASS "WineBottle".

```

### 18.1.7 FileRW Class

The class `FileRW` has only one method, `XternOrder`. This method writes the order to a flat file. It logically belongs in the `Order` object but has been kept separate for operational reasons. A convincing argument can be made to encapsulate file operations in a separate class, much like the view class. If we elect in the future to use a more sophisticated means of externalizing the object, such as a database, we would only have to "plug in" a new `FileRW` class.

```

process pgmname(mixed) test
  IDENTIFICATION DIVISION.
  CLASS-ID. "FileRW" INHERITS SOMObject.
  ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  REPOSITORY.
    CLASS SOMObject          IS "SOMObject"
    CLASS WineOrder          IS "WineOrder".
  DATA DIVISION.
  PROCEDURE DIVISION.

```

#### 1. XternOrder

```

  IDENTIFICATION DIVISION.
  METHOD-ID. "XternOrder".
  ENVIRONMENT DIVISION.
  INPUT-OUTPUT SECTION.
  FILE-CONTROL.
    SELECT ORDERS          ASSIGN TO    ORDERS
    FILE STATUS IS WS-STATUS-FLAG
    ORGANIZATION IS LINE SEQUENTIAL.
  DATA DIVISION.
  FILE SECTION.
  FD ORDERS EXTERNAL
    RECORD CONTAINS 255.
  01 ORDER-RECORD          PIC X(255).
  WORKING-STORAGE SECTION.
  01 WS-STATUS-FLAG        PIC XX.
  01 WS-ORDER-RECORD.
    05 WS-ORDER-NUMBER     PIC X(5).
    05 WS-ORDER-DATE       PIC X(8).
    05 FILLER              PIC XXX.
    05 WS-ITEMS.
      10 WS-ORDER-COUNT    PIC S9(4).
      10 WS-ORDER-ITEM     OCCURS 1 TO 64
                           DEPENDING ON WS-ORDER-COUNT
                           INDEXED BY WS-INDEX.
      15 WSO-TYPE          PIC X(20).
      15 WSO-COST          PIC 999V99.
  LINKAGE SECTION.
  01 orderObj              USAGE OBJECT REFERENCE WineOrder.

```

The method is invoked with the order object passed as a parameter.

```

  PROCEDURE DIVISION      USING orderObj.
    OPEN OUTPUT ORDERS.
    MOVE SPACES TO WS-ORDER-RECORD
    INVOKE orderObj "GetOrderNumber" RETURNING WS-ORDER-NUMBER.
    INVOKE orderObj "GetOrderDate"   RETURNING WS-ORDER-DATE.
    INVOKE orderObj "DescribeOrder"  RETURNING WS-ITEMS.
    WRITE ORDER-RECORD FROM WS-ORDER-RECORD.
    CLOSE ORDERS.
    EXIT METHOD.
  END METHOD "XternOrder".
  END CLASS "FileRW".

```

The object is interrogated to obtain all its data. Note how we reuse the DescribeOrder method in the order object. The first 255 bytes of data are written to a flat file. Obviously, in a more rigorous application, we would use a record long enough to hold all 64 possible entries in the order.

Now we have a working application modelling a subset of the business process functionality. However, more work still needs to be done. We need to extend the application to come closer to the user's requirements.

---

## Chapter 19. The Third Iteration

Further analysis and investigation indicate that we need to extend our application to more accurately model the problem domain.

The first step in our extension of the application is to revisit the use case step. We first add to the original use case, and then create a new use case.

**Extending the Original Use Case:** It is possible that one or all of a customer's selections may not be in stock at the time of the order. When this occurs, the available bottles are added to the order, and the order is set aside to be completed when the out of stock selections arrive in the store's inventory.

**A New Use Case - The Salesperson Checks Old Orders:** When the salesperson checks old or existing orders, he gets the order number from the customer or the order list and retrieves the sales record for the order. The inventory status of selections that were out of stock when the order was filled is verified. This information is conveyed to the salesperson.

From the second use case, we observe that we need a mechanism for checking old orders. We shall create an old order object. It will inherit the attributes and methods of the original wine order class. It will also include methods for checking the inventory status of bottles that were out of stock when the order was taken and a method to retrieve old orders from an external file. We need not retrieve the objects we externalized in the first iteration, so to accommodate this new requirement, we must develop a method to read the orders and place them into objects.

To instantiate an old order object after it is read from our external flat file, we need a method to set all the instance data of the order. To write an image of the object to the flat file, we also need a method to obtain all the instance data. These methods need to be added to our order class. Since we want these methods to be usable across both old and existing order classes, we add them to the existing wineorder class.

In the first iteration, we created a bottle class. We would expect a bottle object to be able to tell us if it is in or out of stock. Therefore, we shall add a `GetStockStatus` method to our bottle class. The `OldOrder` class will invoke this method when it checks on the status of the bottles in the order.

To report the additional requirements of the extensions to the wineorder object and the new objects, we need to add methods to the user interface object. Its new methods will provide an order number, and inform the salesperson of the status of out-of-stock items. We shall also add code to handle the additional commands of status (used to check the order status) and new (used to create new orders).

In summary, our classes now look like the following as illustrated in Tables 21, 22, 23, 24, 25, 26, 27, 28, and 29:

- WineOrder Object

- Attributes:

<i>Table 21. WineOrder Class Attributes for Third Iteration</i>
Order number
Order date
Order contents (bottles)

- Methods:

<i>Table 22. WineOrder Class Methods for Third Iteration</i>	
Method	Purpose
Constructor	Creates an instance of the object.
Destructor	Destroys an instance of the object.
AddBottle	Adds a bottle to the order.
Removebottle	Deletes a bottle from the order.
CalculateCost	Computes the cost of the bottles in the order.
DescribeOrder	Describes the contents of the order.
GetOrderNumber	Returns the order number of an order object.
GetOrderDate	Returns the order date of an order object.
SetOrderNumber	Sets the order number of an order object.
SetOrderDate	Sets the order date of an order object.
SetInstanceData	Sets the attributes of the wineorder object.
GetInstanceData	Returns the attributes of the wineorder object.

- CRC card:

-----	
Class: WineOrder	
-----	
Responsibilities:	Collaborators:
Add a bottle to the order	UIInterface
Remove a bottle from the order	UIInterface
Calculate the cost of order	WineBottle, UIInterface
Describe the contents of an order	WineBottle, UIInterface
Get the order number	UIInterface
Get the order date	UIInterface
Set the order number	UIInterface
Set the order date	UIInterface
Externalize the order	WineBottle
Set all the instance data for an order	FileRW
Get all the instance data for an order	None
Get the stock status of a selection	WineBottle

Figure 51. CRC Card for WineOrder Class in Third Iteration

- OldOrder Object
  - Attributes:

Table 23. OldOrder Class Attributes for Third Iteration
Inherited from WineOrder.

- Methods (inherited from WineOrder):

Table 24. OldOrder Class Methods for Third Iteration	
Method	Purpose
Constructor	Creates an instance of the object.
Destructor	Destroys an instance of the object.
CheckItems	Checks the stock status of selections in the order.

- CRC card:

-----	
Class: OldOrder	
-----	
Responsibilities:	Collaborators:
Add a bottle to the order	UI
Remove a bottle from the order	UI
Calculate the cost of order	WB, UI
Describe the contents of an order	WB, UI
Get the order number	UI
Get the order date	UI
Set the order number	UI
Set the order date	UI
Externalize the order	WB
Set all the instance data for an order	WO, FRW
Get all the instance data for an order	WO
Get the stock status of a selection	WB
Check the stock status of a selection	WB
Read the order data from the order file	FRW

Figure 52. CRC Card for OldOrder Class in Third Iteration

- WineBottle Object

- Attributes:

<i>Table 25. WineBottle Class Attributes for Third Iteration</i>
Cost
Wine type

- Methods:

<i>Table 26. WineBottle Class Methods for Third Iteration</i>	
Method	Purpose
Constructor	Creates an instance of the object.
Destructor	Destroys an instance of the object.
GetCost	Returns the cost of the bottle.
GetType	Returns the type of wine in the bottle.
SetCost	Sets the cost of the bottle.
SetType	Sets the type of wine in the bottle.
GetStatus	Returns the inventory status of the bottle.



- CRC card:

-----	
Class: WineBottle	
-----	
Responsibilities:	Collaborators:
Get the bottle cost	WineOrder
Get the wine type	WineOrder
Set the bottle cost	WineOrder
Set the wine type	WineOrder
Get the stock status	WineOrder, OldOrder

Figure 53. CRC Card for WineBottle Class in Third Iteration

- FileRW Object
  - Attributes: (none)
  - Methods:

Table 27. FileRW Class Methods for Third Iteration	
Method	Purpose
Constructor	Creates an instance of the object.
Destructor	Destroys an instance of the object.
XternOrder	Writes the object to a flat file.
XReadOrder	Reads the object from the flat file.

- CRC card:

-----	
Class: FileRW	
-----	
Responsibilities:	Collaborators:
Externalize the order to a file	WineOrder
Reads an order from the file	OldOrder

Figure 54. CRC Card for FileRW Class in Third Iteration

- UserInterface Object
 

To isolate the business logic from the presentation logic, we add an object that interfaces with the user of the application.

  - Attributes:

Table 28. UserInterface Class Attributes for Third Iteration	
Action	
Bottle (selected)	

- Methods:

Table 29. <i>UserInterface Class Methods for Third Iteration</i>	
Method	Purpose
Constructor	Creates an instance of the object.
Destructor	Destroys an instance of the object.
ReadAction	Gets the input command from the system user.
ReadType	Gets the type of wine from the system user.
ReadCost	Gets the cost of the bottle from the system user.
ReadProcess	Gets the input command from the system user.
ReadOrder	Gets the order number from the system user.
WriteMessage	Displays a system status message to the user.
WriteOutput	Displays the cost of the order and order number to the user.
WriteBottle	Displays the attributes of a bottle collected in the order to user.
WriteStatus	Displays the out-of-stock selections to the user.

- CRC card:

-----	
Class: UserInterface	
-----	
Responsibilities:	Collaborators:
Accept a request from the system user:	
--Add a bottle to order	WineOrder, WineBottle
--Remove a bottle from order	WineOrder, WineBottle
--List contents of order	WineOrder, WineBottle
--Locate order using order number	WineOrder
Respond to the system user:	
--Display a status message	WineOrder
--Display the order cost and the order number	WineOrder, WineBottle
--Display the order contents	WineOrder, WineBottle

Figure 55. *CRC Card for UserInterface Class in Third Iteration*

### 19.1.1 Object Interaction Diagram

To see how our objects will interact with each other, we can create a diagram which shows the flow from one object to another for the business process. An object interaction diagram for this iteration of the development effort follows as shown in Figure 56.

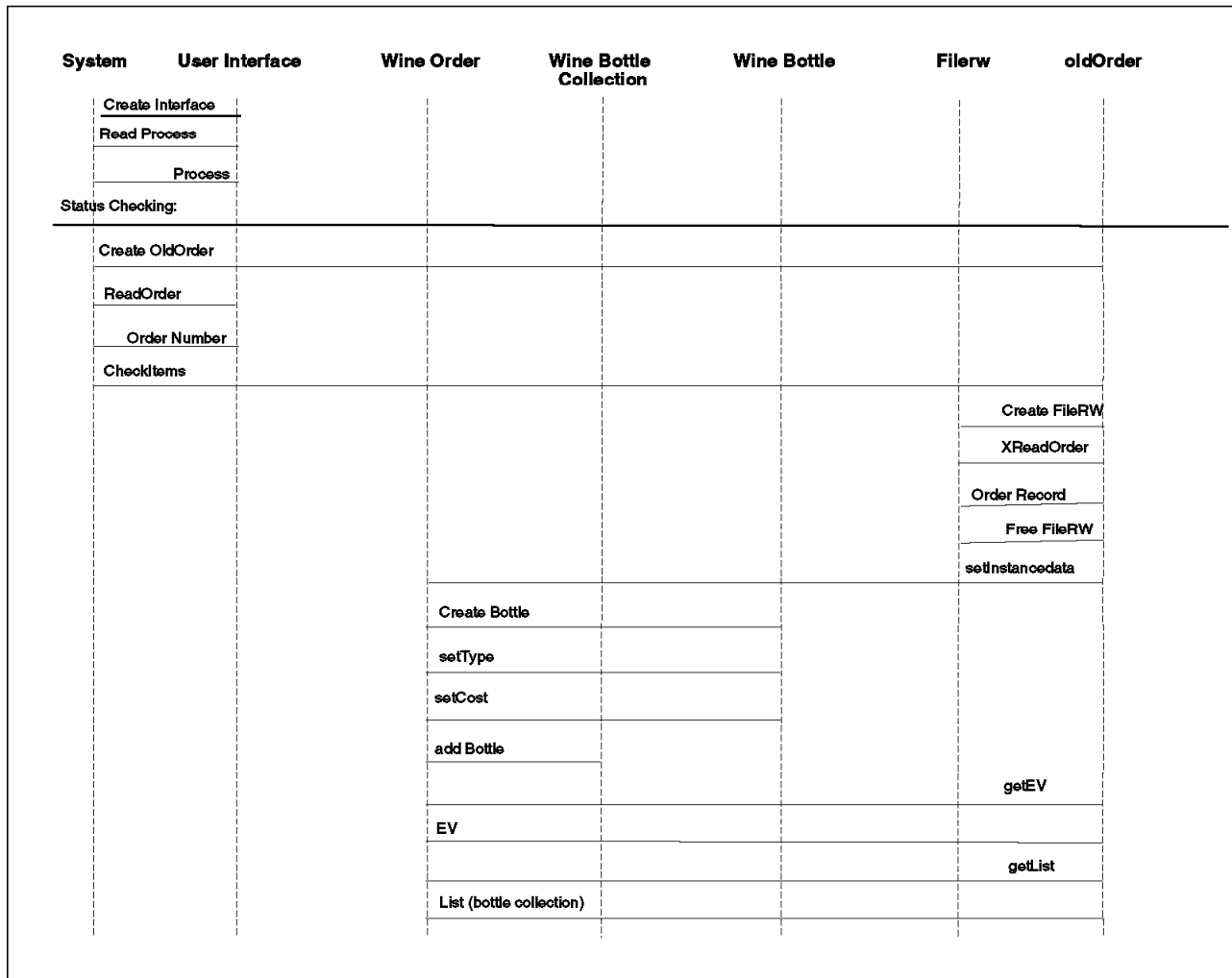


Figure 56. Object Interaction Diagram – Third Iteration

## 19.2 Code Design

For the third phase, we implement the SetInstanceData and GetInstanceData methods for the existing WineOrder object. We also implement the OldOrder object inheriting from the WineOrder object with the addition of the CheckBott method. Further, we extend the WineBottle class with the GetStatus method. The FileRW class must have the XReadOrder method added to it. We extend the UserInterface class by the addition of the ReadProcess, ReadOrder, and WriteStatus methods.

Also, we extend our client program to use the new classes and methods, and add the logic to handle the new commands.

We start by adding the new methods to our existing classes, then create the new class, and finally, extend our client program.

*Adding GetStatus to WineBottle:* As a practical point, and to confine the scope of the exercise, we generate a random integer and divide it by a constant, then check the remainder. If it is evenly divisible we define it as out of stock; otherwise, it is in-stock. For this iteration, we use the integer 2 as the divisor,

providing a fifty percent chance of the item being in or out of stock. In a real application, we would verify the stock status from the number on hand indicated in an inventory record.

*The OldOrder Class:* The OldOrder class is straightforward. It creates a filerw object, and invokes it to read the data from the file that was written at the end of the ordering process. It also invokes the SetInstanceData method it inherited from the WineOrder class to inflate the object. The CheckItem method invokes the GetInstanceData from the WineOrder class, then invokes the GetStatus method of the WineBottle class for each bottle of the order. The out-of-stock items are placed in a table returned to the invoker. In this case, it is the Wine client program.

Since we are using SOM collection classes we must include methods in the order object to retrieve the environment variable, the iterator, and the collection created during the initialization of the order object. When we instantiate the oldorder object, an order object is also created, and the environment variable, iterator, and collection are built.

*The WineOrder Class:* For this iteration, we add two methods, SetInstanceData and GetInstanceData, to the WineOrder class. The purpose is to inflate objects or retrieve their attributes in one single method invocation.

The SetInstanceData uses the AddBottle method contained in the class. The method must pass through the table of ordered items. For each one, it creates a WineBottle object, and then invokes the AddBottle method on itself to create the collection of items in the order.

The GetInstanceData performs the same operation in reverse. That is, it collects and builds a table of the items in the collection. It also invokes the GetOrderNumber and GetOrderDate methods to obtain these attributes of the order.

To accommodate the oldorder object's requirement to process the collection, we write methods to retrieve the SOM environment variable, iterator, and collection in the order class. These are called GetEV, GetList, and GetIterator.

*The UserInterface Class:* The changes required to this class are easily implemented. The ReadProcess method obtains the desired process from the system user and is implemented like the ReadAction method coded earlier. The WriteStatus method expects a table of out-of-stock items and passes through this table displaying them to the user. ReadOrder is also easy to implement.

*The FileRW Class:* This class is extended by the addition of the XReadOrder method, which performs the inverse of the XternOrder method. It reads the file until a record for the requested order number is found, and returns a structure containing the record to the invoker, oldorder. By allowing oldorder to inflate the object, FileRW does not need the handle of the order object.

*The Wine Client Program:* Finally, we must modify the Wine client program that drives our application process. This program must be extended to process the new and status commands. For the new command, we put the existing code in a paragraph and perform it.

For the status command we must draft additional logic. We invoke the UserInterface object to get the order number to be checked; once that data is

obtained, we invoke the OldOrder object to read the flat file and inflate the order object. Then the OldOrder object is invoked to check the status of the ordered items, and the resulting table passed to the UserInterface object for display to the system user.

This iteration makes a number of changes. We should use a debugger to ensure that the flow of the application is correct, and that all the affected interfaces are working properly.

---

## 19.3 Code Commentary

This section contains extracts of the important features of the client and class programs and descriptions of how the code works.

### 19.3.1 Wine Client

We update the wine program to offer the user a new option, checking and updating the status of an order. We also include a minor enhancement to the order date and order number routines.

```
process pgmname(longmixed) test
  IDENTIFICATION DIVISION.
  PROGRAM-ID. "Wine".
  ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  REPOSITORY.
    CLASS SOMObject          IS "SOMObject"
    CLASS WineOrder         IS "WineOrder"
    CLASS OldOrder          IS "OldOrder"
    CLASS Bottle            IS "WineBottle"
    CLASS FileRW            IS "FileRW"
    CLASS UserInterface     IS "UserInterface".
```

The new class of OldOrder is added to the list of referenced classes.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 orderObj          USAGE OBJECT REFERENCE WineOrder.
01 oldOrderObj       USAGE OBJECT REFERENCE OldOrder.
01 userObj           USAGE OBJECT REFERENCE UserInterface.
01 bottleObj         USAGE OBJECT REFERENCE Bottle.
01 fileObj           USAGE OBJECT REFERENCE FileRW.
```

Again, the new class OldOrder is referenced.

```
01 ACTION            PIC X(10).
01 PROCESS           PIC X(10).
```

These items are connected with the new interface. This offers the user an extra option, checking the status.

```
01 OUT-ITEMS.
  05 OUT-COUNT       PIC S9(4).
  05 OUT-ITEM        OCCURS 1 TO 64 TIMES
                     DEPENDING ON OUT-COUNT
                     INDEXED BY OUT-INDEX.
  10 OUT-TYPE        PIC X(20).
  10 OUT-COST        PIC 999V99.
```

```

PROCEDURE DIVISION.
    INVOKE UserInterface "somNew" RETURNING userObj.
    INVOKE userObj "ReadProcess" RETURNING PROCESS.
    PERFORM UNTIL PROCESS (1:4) = "EXIT"
        EVALUATE PROCESS (1:3)
            WHEN "STA"
                PERFORM CHECK-OLD-ORDER THRU CHECK-EXIT
            WHEN "NEW"
                PERFORM CREATE-NEW-ORDER THRU CREATE-EXIT
            WHEN OTHER
                CONTINUE
        END-EVALUATE
    INVOKE userObj "ReadProcess" RETURNING PROCESS
    END-PERFORM.

```

In this iteration the user can check an order's status. This code also cleans up and exits.

```

    INVOKE userObj "somFree".
    GOBACK.

CHECK-OLD-ORDER.
    INVOKE oldOrder "somNew" RETURNING oldOrderObj.
    INVOKE userObj "ReadOrder" RETURNING ORDER-NUMBER.
    INVOKE oldOrderObj "CheckItems" USING ORDER-NUMBER
        RETURNING OUT-ITEMS.

```

We create an oldorder object and invoke the user interface object to get the order number. Then we pass the order number to the CheckItems method of the oldorder object, which returns a structure containing the out-of-stock items.

```

    INVOKE userObj "WriteStatus" USING OUT-ITEMS.
    INVOKE oldOrderObj "somFree".
CHECK-EXIT.
EXIT.

```

We invoke the user interface object to display the list of out-of-stock items to the user, and destroy the oldorder object created during the process.

```

CREATE-NEW-ORDER.

```

We take the code written in the client program in the second iteration and place it in a paragraph, which is performed from the code processing the command.

```

    MOVE FUNCTION CURRENT-DATE TO ORDER-DATE.
    COMPUTE WS-RANDOM-VAL = FUNCTION RANDOM.
    COMPUTE ORDER-NUMBER = WS-RANDOM-VAL * 10000.

```

This improves on the random-number use in version one.

```

    MOVE ZERO to ITEM-COUNT

```

This initializes the counter for the number of items ordered. Since it is possible to loop through this section of code, an initialization is required to allow multiple orders to be placed during one execution of the client program.

The remainder of the code is a repetition of the code in version two.

```

.
.
.
CREATE-EXIT.
EXIT.
END PROGRAM "Wine".

```

### 19.3.2 WineOrder Class

The WineOrder object is changed as follows:

```

process pgmname(mixed) test
  IDENTIFICATION DIVISION.
  CLASS-ID.  "WineOrder" INHERITS SOMObject.
  ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  REPOSITORY.
    CLASS SOMObject          IS "SOMObject"
    CLASS SOMCollection      IS "somf_TSet"
    CLASS SOMIterator        IS "somf_TSetIterator"
    CLASS WineBottle         IS "WineBottle".

```

The Identification Division, the Data Division (not shown here), and the Environment Divisions are the same as in Version Two.

PROCEDURE DIVISION.

#### 1. GetEV

This method and the next two are added to allow the new method Checkitems to work. Checkitems is not defined here but in a subclass (OldOrder) so these three methods can be defined there too. However, by placing them in the superclass, and other classes subsequently defined as subclasses of this we can inherit these methods. Note the use of set verbs instead of move. In actuality, we pass pointer values, not data. So, you cannot just move an object reference, you have to set it.

```

  IDENTIFICATION DIVISION.
  METHOD-ID. "GetEV".
  DATA DIVISION.
  WORKING-STORAGE SECTION.
  LINKAGE SECTION.
  01 LS-EV                      USAGE POINTER.
  PROCEDURE DIVISION
    RETURNING  LS-EV.
    SET LS-EV TO WS-EV.
    EXIT METHOD.
  END METHOD "GetEV".

```

#### 2. GetList

See the comment for methods "GetEV" above.

```

  IDENTIFICATION DIVISION.
  METHOD-ID. "GetList".
  DATA DIVISION.
  WORKING-STORAGE SECTION.
  LINKAGE SECTION.
  01 LS-LIST                    USAGE OBJECT REFERENCE SOMCollection.
  PROCEDURE DIVISION
    RETURNING  LS-LIST.
    SET LS-LIST TO WINEORDER-LIST.
    EXIT METHOD.
  END METHOD "GetList".

```

### 3. GetIterator

See the comment for methods "GetEV" above.

```
IDENTIFICATION DIVISION.  
METHOD-ID. "GetIterator".  
DATA DIVISION.  
WORKING-STORAGE SECTION.  
LINKAGE SECTION.  
01 LS-ITERATOR USAGE OBJECT REFERENCE SOMIterator.  
PROCEDURE DIVISION          RETURNING LS-ITERATOR.  
    SET LS-ITERATOR TO WINEORDER-ITERATOR.  
    EXIT METHOD.  
END METHOD "GetIterator".
```

### 4. GetOrderNumber

This is the same as Versions One and Two.

### 5. GetOrderDate

This is the same as Versions One and Two.

### 6. SetOrderNumber

This is the same as Versions One and Two.

### 7. SetOrderDate

This is the same as Versions One and Two.

### 8. DescribeOrder

This is the same as Versions One and Two.

### 9. CalculateCost

This is the same as Versions One and Two.

### 10. AddBottle

This is the same as Versions One and Two.

### 11. RemoveBottle

This is the same as Versions One and Two.

### 12. SetInstanceData

```
IDENTIFICATION DIVISION.  
METHOD-ID. "SetInstanceData".  
DATA DIVISION.  
WORKING-STORAGE SECTION.  
01 WS-PARMS.  
    05 ITEM-COUNT          PIC S9(4)      COMP.  
    05 WS-FLAG             PIC X.  
        88 SUCCESSFUL          VALUE "0".  
        88 FAILURE             VALUE "1".  
01 bottleObj              USAGE OBJECT REFERENCE WineBottle.  
LINKAGE SECTION.  
01 LS-ORDER.  
    05 LS-ORDER-NUMBER     PIC X(5).  
    05 LS-ORDER-DATE       PIC X(8).  
    05 FILLER              PIC XXX.  
    05 LS-ORDER-COUNT      PIC S9(4).  
    05 LS-ORDER-ITEM       OCCURS 1 TO 64 TIMES  
                           DEPENDING ON LS-ORDER-COUNT.  
    10 LSO-TYPE            PIC X(20).
```



```

        10  LSO-COST                PIC 999V99.
PROCEDURE DIVISION
        USING      LS-ORDER.
        INVOKE SELF "SetOrderNumber" USING LS-ORDER-NUMBER.
        INVOKE SELF "SetOrderDate"   USING LS-ORDER-DATE.
        PERFORM VARYING LS-INDEX FROM 1 BY 1
            UNTIL  LS-INDEX > LS-ORDER-COUNT
                INVOKE WineBottle "somNew" RETURNING bottleObj
                INVOKE bottleObj  "SetType" USING LSO-TYPE (LS-INDEX)
                INVOKE bottleObj  "SetCost" USING LSO-COST (LS-INDEX)
                INVOKE SELF "AddBottle" USING bottleObj
                                RETURNING WS-PARMS

        END-PERFORM.
        EXIT METHOD.
    END METHOD "SetInstanceData".

```

Keep in mind that we have an empty order object, and we are building it with attributes. We are also building the collection of wine bottles in the order from the structure contained in the record from the flat file. This process is known as inflating an object.

### 13. GetInstanceData

```

IDENTIFICATION DIVISION.
METHOD-ID. "GetInstanceData".
DATA DIVISION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
01  LS-ORDER.
    05  LS-ORDER-NUMBER          PIC X(5).
    05  LS-ORDER-DATE           PIC X(8).
    05  FILLER                  PIC XXX.
    05  LS-ITEMS.
        10  LS-ORDER-COUNT      PIC S9(4).
        10  LS-ORDER-ITEM      OCCURS 1 TO 64 TIMES
                                DEPENDING ON LS-ORDER-COUNT.
            15  LSO-TYPE        PIC X(20).
            15  LSO-COST        PIC 999V99.
PROCEDURE DIVISION
        RETURNING  LS-ORDER.
        INVOKE SELF "GetOrderNumber" RETURNING LS-ORDER-NUMBER.
        INVOKE SELF "GetOrderDate"   RETURNING LS-ORDER-DATE.
        INVOKE SELF "DescribeOrder"  RETURNING LS-ITEMS.
        EXIT METHOD.
    END METHOD "GetInstanceData".

```

This method copies all the attributes of the object to a record area to be written to a flat file. This process is known as flattening an object. Note the reuse of the "DescribeOrder" method.

### 14. somDefaultInit

This is the same as Version Two

### 15. somFree

This is the same as Version Two

```
END CLASS "WineOrder".
```

### 19.3.3 OldOrder Class

This is the completely new class in Version Three. It inherits from a class that has already been used: WineOrder. This means that any method that could be invoked against an object of WineOrder can be invoked against an object of this class.

For example, it was possible to invoke the method AddBottle against an object of the WineOrder class. Therefore, it is possible to invoke it against an object of this class, OldOrder without having to repeat the method's definition.

```
process test pgmname(longmixed)
  IDENTIFICATION DIVISION.
  CLASS-ID. "OldOrder" INHERITS WineOrder.
```

The CLASS-ID statement specifies from which class this class directly inherits. Previously the ancestor class was almost always SOMObject.

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
  CLASS OldOrder          IS "OldOrder"
  CLASS WineOrder         IS "WineOrder"
  CLASS WineBottle        IS "WineBottle"
  CLASS FileRW            IS "FileRW"
  CLASS SOMCollection      IS "sopf_TSet"
  CLASS SOMIterator       IS "sopf_TSetIterator".

DATA DIVISION.
```

No data items are defined because all the attributes of this class have already been defined in WineOrder. If, for example, a program wished to discover the order number of an object of this class it would invoke the method GetOrderNumber against the object and the method would return the value. For a method defined in this class to find out the order-number, it too would issue the GetOrderNumber method. Only this time, the object-handle would be replaced by the keyword SELF. For example,

```
  Invoke objhandle "GetOrderNumber" etc
becomes
  Invoke SELF "GetOrderNumber" etc
```

Below are actual examples in the code for method CheckItems.

#### 1. CheckItems

```
IDENTIFICATION DIVISION.
METHOD-ID. "CheckItems".
DATA DIVISION.
WORKING-STORAGE SECTION.
01 CollectedBottle      USAGE OBJECT REFERENCE WineBottle.
01 WINEORDER-LIST        USAGE OBJECT REFERENCE SOMCollection.
01 WINEORDER-ITERATOR    USAGE OBJECT REFERENCE SOMIterator.
01 fileObj              USAGE OBJECT REFERENCE FileRW.
01 WS-FLAG               PIC X.
   88 OUT-OF-STOCK       VALUE "0".
   88 IN-STOCK           VALUE "1".
01 WS-TYPE               PIC X(20).
01 WS-COST               PIC 999V99.
01 WS-EV                USAGE POINTER.
01 ITEM-COUNT            PIC S9(8) COMP.
```

```

01 WS-ORDER-RECORD.
   05 WSO-ORDER-NUMBER      PIC X(5).
   05 WSO-ORDER-DATE        PIC X(8).
   05 FILLER                 PIC XXX.
   05 WSO-ITEMS.
      10 WSO-ORDER-COUNT    PIC S9(4).
      10 WSO-ORDER-ITEM OCCURS 1 TO 64
                           DEPENDING ON WSO-ORDER-COUNT
                           INDEXED BY WSO-INDEX.
      10 WSOR-TYPE          PIC X(20).
      10 WSOR-COST          PIC 999V99.
LINKAGE SECTION.
01 orderObj                 USAGE OBJECT REFERENCE WineOrder.
01 LS-OUT-ITEMS.
   05 LS-OUT-COUNT          PIC S9(4)   COMP-5.
   05 LS-OUT-ITEM OCCURS 1 TO 64
                           DEPENDING ON LS-OUT-COUNT
                           INDEXED BY LS-INDEX.
      10 LSO-TYPE           PIC X(20).
      10 LSO-COST           PIC 999V99.

PROCEDURE DIVISION
      USING orderObj
      RETURNING LS-OUT-ITEMS.
      INVOKE FileRW "somNew" RETURNING fileObj.
      INVOKE fileObj "XReadOrder" USING LS-ORDER-NUMBER
                                RETURNING WS-ORDER-RECORD.
      INVOKE fileObj "somFree".

```

This code uses the XReadOrder method of the filerw object to retrieve the order record from the flat file.

```

      INVOKE self "SetInstanceData" USING WS-ORDER-RECORD.

```

We use the record to inflate our order object with the SetInstanceData method. This is inherited from the order object, hence, the invocation on "self."

```

      INVOKE self "GetEV" RETURNING WS-EV.
      INVOKE self "GetList" RETURNING WINEORDER-LIST.
      INVOKE self "GetIterator"
                                RETURNING WINEORDER-ITERATOR.

```

This provides the data to process the collection created when we inflated our order object.

```

      INVOKE WINEORDER-LIST "somfCount"
                                USING BY VALUE WS-EV
                                RETURNING ITEM-COUNT.
      MOVE ZERO TO LS-OUT-COUNT.
      IF ITEM-COUNT > 0
      THEN INVOKE WINEORDER-ITERATOR "somfFirst"
                                USING BY VALUE WS-EV
                                RETURNING CollectedBottle
      PERFORM CHECK-STATUS
      END-IF.

```

This code operates on the first object in the collection. The next, identical, portion operates on the remaining objects.

```

SUBTRACT 1 FROM ITEM-COUNT.
IF ITEM-COUNT > 0
    THEN PERFORM ITEM-COUNT TIMES
        INVOKE WINEORDER-ITERATOR "somfNext"
            USING      BY VALUE WS-EV
            RETURNING CollectedBottle
        PERFORM CHECK-STATUS
    END-PERFORM
END-IF.
EXIT METHOD.

```

The performed code follows.

```

CHECK-STATUS.
    INVOKE CollectedBottle "GetStatus"
        RETURNING WS-FLAG.
IF OUT-OF-STOCK
    THEN ADD 1 TO LS-OUT-COUNT
        INVOKE CollectedBottle "GetType"
            RETURNING WS-TYPE
        MOVE WS-TYPE TO LSO-TYPE (LS-INDEX)
        INVOKE CollectedBottle "GetCost"
            RETURNING WS-COST
        MOVE WS-COST TO LSO-COST (LS-INDEX)
        SET LS-INDEX UP BY 1.

```

The method concludes:

```

END METHOD "CheckItems".

```

## 2. somFree

```

IDENTIFICATION DIVISION.
METHOD-ID. "somFree"          OVERRIDE.
DATA DIVISION.
WORKING-STORAGE SECTION.
PROCEDURE DIVISION.
    INVOKE SUPER "somFree".
END METHOD "somFree".

```

This is an overridden class that invokes the destructor on the parent class. somFree is invoked on super.

```

END CLASS "OldOrder".

```

## 19.3.4 UserInterface Class

The UserInterface class has three new methods added to it: ReadProcess, ReadOrder, and WriteStatus.

```

process pgmname(mixed) test
    IDENTIFICATION DIVISION.
    CLASS-ID. "UserInterface"    INHERITS SOMObject.
    ENVIRONMENT DIVISION.
    CONFIGURATION SECTION.
    REPOSITORY.
        CLASS SOMObject        IS "SOMObject".
    DATA DIVISION.
    WORKING-STORAGE SECTION.
    01 USER-ACTION                PIC X(10).
        88 UA-ADD                VALUE "Add".
        88 UA-DELETE            VALUE "Delete".
        88 UA-END                VALUE "End".
        88 UA-NEW                VALUE "New".

```

88	UA-STATUS	VALUE "Status".
88	UA-EXIT	VALUE "Exit".

The new inputs that the user can make are listed below.

#### PROCEDURE DIVISION.

##### 1. ReadAction

This is the same as Version Two.

##### 2. ReadType

This is the same as Version Two.

##### 3. ReadCost

This is the same as Version Two.

##### 4. WriteMessage

This is the same as Version Two.

##### 5. WriteOutput

This is the same as Version Two.

##### 6. WriteBottle

This is the same as Version Two.

##### 7. ReadProcess

IDENTIFICATION DIVISION.

METHOD-ID. "ReadProcess".

DATA DIVISION.

WORKING-STORAGE SECTION.

01 WS-EDIT-FLAG PIC X.

LINKAGE SECTION.

01 LS-PROCESS PIC X(10).

PROCEDURE DIVISION RETURNING LS-PROCESS.

MOVE LOW-VALUE TO WS-EDIT-FLAG.

PERFORM UNTIL WS-EDIT-FLAG NOT = LOW-VALUE

DISPLAY "Enter process desired: new, status: "

ACCEPT USER-ACTION FROM SYSIN

MOVE FUNCTION UPPER-CASE (USER-ACTION) TO USER-ACTION

MOVE USER-ACTION TO LS-PROCESS

EVALUATE USER-ACTION (1:3)

WHEN "NEW"

MOVE HIGH-VALUE TO WS-EDIT-FLAG

WHEN "STA"

MOVE HIGH-VALUE TO WS-EDIT-FLAG

WHEN OTHER

DISPLAY "Requested process was " USER-ACTION

DISPLAY "Wrong! Get it right this time!!!"

END-EVALUATE

END-PERFORM.

EXIT METHOD.

END METHOD "ReadProcess".

##### 8. ReadOrder

```

IDENTIFICATION DIVISION.
METHOD-ID. "ReadOrder".
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-EDIT-FLAG          PIC X.
01 WS-ORDER              PIC X(5).
01 WS-ORDER-9            PIC 9(5).
LINKAGE SECTION.
01 LS-ORDER              PIC X(5).
PROCEDURE DIVISION      RETURNING  LS-ORDER.
    MOVE LOW-VALUE TO WS-EDIT-FLAG.
    PERFORM UNTIL WS-EDIT-FLAG = HIGH-VALUE
        DISPLAY "Enter the order number: "
        ACCEPT WS-ORDER          FROM SYSIN
        COMPUTE WS-ORDER-9 = FUNCTION NUMVAL (WS-ORDER)
        MOVE WS-ORDER-9 TO LS-ORDER
        IF LS-ORDER NUMERIC
            THEN MOVE HIGH-VALUE TO WS-EDIT-FLAG
        ELSE
            DISPLAY "Order number is not numeric - try again "
            DISPLAY "and get it right this time!!! "
        END-IF
    END-PERFORM.
    EXIT METHOD.
END METHOD "ReadOrder".

```

#### 9. WriteStatus

```

IDENTIFICATION DIVISION.
METHOD-ID. "WriteStatus".
DATA DIVISION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
01 LS-OUT-ITEMS.
    05 LS-OUT-COUNT          PIC S9(4).
    05 LS-OUT-ITEM          OCCURS 1 to 64 TIMES
                           DEPENDING ON LS-OUT-COUNT
                           INDEXED BY LS-INDEX.
    10 LSO-TYPE              PIC X(20).
    10 LSO-COST              PIC 999V99.
PROCEDURE DIVISION      USING  LS-OUT-ITEMS.
    IF LS-OUT-COUNT > 0
        THEN DISPLAY "LIST OUT OF STOCK ITEMS: "
            INVOKE SELF "WriteBottle"
                USING LS-OUT-ITEMS
    ELSE
        DISPLAY "ALL ITEMS IN STOCK!".
    EXIT METHOD.
END METHOD "WriteStatus".

```

In the description of the class `OldOrder` comments exist about the use of `Invoke SELF` in the context of a class working with attributes defined in its superclass. This use of `Invoke SELF`, however, is to use a method defined elsewhere in the same class.

```

    ELSE
        DISPLAY "ALL ITEMS IN STOCK!".
    EXIT METHOD.
END METHOD "WriteStatus".
END CLASS "UserInterface".

```

### 19.3.5 Bottle Class

In Version Three, WineBottle gains a method, GetStatus.

```
process pgmname(mixed) test
  IDENTIFICATION DIVISION.
  CLASS-ID.  "WineBottle"    INHERITS somf-MCollectible.
  ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  REPOSITORY.
    CLASS WineBottle          IS "WineBottle"
    CLASS somf-MCollectible   IS "somf_MCollectible".
  DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 WINEBOTTLE-OBJECT.
    05 WINE-TYPE              PIC X(20).
    05 WINE-COST              PIC 999V99.
```

All three initial divisions are as in Version 2.

PROCEDURE DIVISION.

1. somflsEqual

This is the same as Version Two.

2. GetCost

This is the same as Version Two.

3. SetCost

This is the same as Version Two.

4. GetType

This is the same as Version Two.

5. SetType

This is the same as Version Two.

6. GetStatus

```
  IDENTIFICATION DIVISION.
  METHOD-ID. "GetStatus".
  DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 WS-STATUS-WORK          PIC 9(5).
  01 WS-STATUS-MOD          PIC 9.
  01 WS-RANDOM-WORK         PIC 9V9(5).
  LINKAGE SECTION.
  01 LS-STATUS              PIC X.
  PROCEDURE DIVISION
    RETURNING LS-STATUS.
    COMPUTE WS-RANDOM-WORK = FUNCTION RANDOM.
    COMPUTE WS-STATUS-WORK = WS-RANDOM-WORK * 10000.
    DIVIDE WS-STATUS-WORK BY 2 GIVING WS-STATUS-WORK
      REMAINDER WS-STATUS-MOD.
```

In actual applications order status is determined by a check against a database or an equivalent action. In our example the Random function decides the status. By using the integer 2 as a divisor, there is a 50% chance of the item being in stock. In actual applications, this would be irritating to the customer.

```

        IF WS-STATUS-MOD = 0
            THEN MOVE "0" TO LS-STATUS
        ELSE
            MOVE "1" TO LS-STATUS.
        EXIT METHOD.
    END METHOD "GetStatus".
END CLASS "WineBottle".

```

### 19.3.6 FileRW Class

Two changes occur for this class compared with Version Two. The XternOrder method writes the order to a file using the new method of the WineOrder class, GtInstanceData. A new method XReadOrder reads the order details from the file.

```

process pgmname(mixed) test
    IDENTIFICATION DIVISION.
    CLASS-ID. "FileRW" INHERITS SOMObject.
    ENVIRONMENT DIVISION.
    CONFIGURATION SECTION.
    REPOSITORY.
        CLASS SOMObject          IS "SOMObject"
        CLASS WineOrder          IS "WineOrder".
    DATA DIVISION.
    WORKING-STORAGE SECTION.

```

There is no difference in the Identification, Environment, or Data Divisions compared with the previous versions.

```

    PROCEDURE DIVISION.

```

#### 1. XternOrder

```

        IDENTIFICATION DIVISION.
        METHOD-ID. "XternOrder".
        ENVIRONMENT DIVISION.
        .
        DATA DIVISION.
        FILE SECTION.
        .
        WORKING-STORAGE SECTION.
        .
        LINKAGE SECTION.
        01 orderObj          USAGE OBJECT REFERENCE WineOrder.
        PROCEDURE DIVISION  USING orderObj.

```

This code is the same as Version Two.

```

        OPEN OUTPUT ORDERS.
        MOVE SPACES TO WS-ORDER-RECORD.
        INVOKE orderObj "GetInstanceData"
            RETURNING WS-ORDER-RECORD.

```

In Version Two, the individual attributes were acquired, but the new method in WineOrder makes it possible to acquire all the data with one method invocation.

```

        WRITE ORDER-RECORD FROM WS-ORDER-RECORD.
        CLOSE ORDERS.
        EXIT METHOD.
    END METHOD "XternOrder".

```

#### 2. XReadOrder



```

IDENTIFICATION DIVISION.
METHOD-ID. "XReadOrder".
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT ORDERS          ASSIGN TO    ORDERS
        FILE STATUS IS WS-STATUS-FLAG
        ORGANIZATION IS LINE SEQUENTIAL.
DATA DIVISION.
FILE SECTION.
FD  ORDERS EXTERNAL
    RECORD CONTAINS 255.
01  ORDER-RECORD          PIC X(255).
WORKING-STORAGE SECTION.
01  WS-STATUS-FLAG        PIC XX.
01  WS-EOF-FLAG           PIC X.
LINKAGE SECTION.
01  LS-ORDER              PIC X(5).
01  LS-ORDER-RECORD.
    05  LS-ORDER-NUMBER    PIC X(5).
    05  LS-ORDER-DATE      PIC X(8).
    05  FILLER             PIC XXX.
    05  LS-ORDER-COUNT     PIC S9(4).
    05  LS-ORDER-ITEM      OCCURS 1 TO 64
                            DEPENDING ON LS-ORDER-COUNT
                            INDEXED BY LS-INDEX.
    10  LSO-TYPE           PIC X(20).
    10  LSO-COST           PIC 999V99.
PROCEDURE DIVISION
    USING LS-ORDER
    RETURNING LS-ORDER-RECORD.

    OPEN INPUT  ORDERS.
    MOVE LOW-VALUE TO WS-EOF-FLAG.
    PERFORM UNTIL WS-EOF-FLAG = HIGH-VALUE
        OR LS-ORDER-NUMBER = LS-ORDER
        READ ORDERS INTO LS-ORDER-RECORD
        AT END MOVE HIGH-VALUE TO WS-EOF-FLAG
        NOT AT END
        IF LS-ORDER-NUMBER = LS-ORDER
            THEN CONTINUE
        END-IF
    END-READ
    END-PERFORM.
    CLOSE ORDERS.
    EXIT METHOD.
    END METHOD "XReadOrder".
END CLASS "FileRW".

```

### 19.3.6.1 Conclusion

After the third iteration of our development, we more closely model the business process of the wine store. As we probe further, we discover that our application isn't quite complete, and another iteration is required.



---

## Chapter 20. The Fourth Iteration

As typically happens in development projects, more requirements came in after coding started. But, in an iterative development process, this is expected and easily handled. The ramifications to our existing code are minimal, an advantage of the object-oriented approach.

To accommodate the new requirement from the users, we must create another use case.

***The Salesperson Checks the Status of Old Orders:*** When the salesperson checks the status of old orders in the system, he reviews the orders in the order file. For orders that have unfilled selections, he checks the inventory for the selected bottles. If they are present, he adds them to the order. During this review, the salesperson also wants to know how many old orders are present in the system.

This use case indicates the need for a metaclass with information about an existing class. To accommodate this need, we create a `MetaOldOrder` object to count the old orders, and create an old order object to be checked.

The classes (`WineOrder` and `WineBottle`) remain unchanged. `OldOrder` inherits from the new metaclass, `MetaOldOrder`. `UserInterface` needs a method added to display the count of orders checked. The client program needs to process an additional command for status.

Further, the previous iteration included a shortcoming – a message indicating that a requested order could not be found on the order file was not produced. This iteration corrects this error.

To avoid redundancy, we do not summarize the existing classes.

Our new class, MetaOldOrder, consists of the following as shown in Tables 30 and 31:

- MetaOldOrder Object
  - Attributes:

<i>Table 30. MetaOldOrder Class Attribute for Fourth Iteration</i>
Status count.

- Methods:

<i>Table 31. MetaOldOrder Class Methods for Fourth Iteration</i>	
Method	Purpose
CreateOldOrder	Creates an old order object.
CountOldOrder	Counts the number of old orders created.

- CRC card:

-----	
Class: MetaOldOrder	
-----	
Responsibilities:	Collaborators:
Create an old order	OldOrder
Count the number of old orders	OldOrder

Figure 57. CRC Card for MetaOldOrder in Fourth Iteration

UserInterface has the following additions as shown in Table 32:

- UserInterface Object
  - Attributes: (unchanged from previous versions)
  - Methods:

<i>Table 32. UserInterface Class Methods for Fourth Iteration</i>	
Method	Purpose
WriteOutCount	Displays the number of old orders checked.
WriteLost	displays a message indicating that the requested order could not be found in the order file.

## 20.1.1 Object Interaction Diagram

To see how our objects will interact with each other, we can create a diagram which shows the flow from one object to another for the business process. An object interaction diagram for this iteration of the development effort follows as shown in Figure 58.

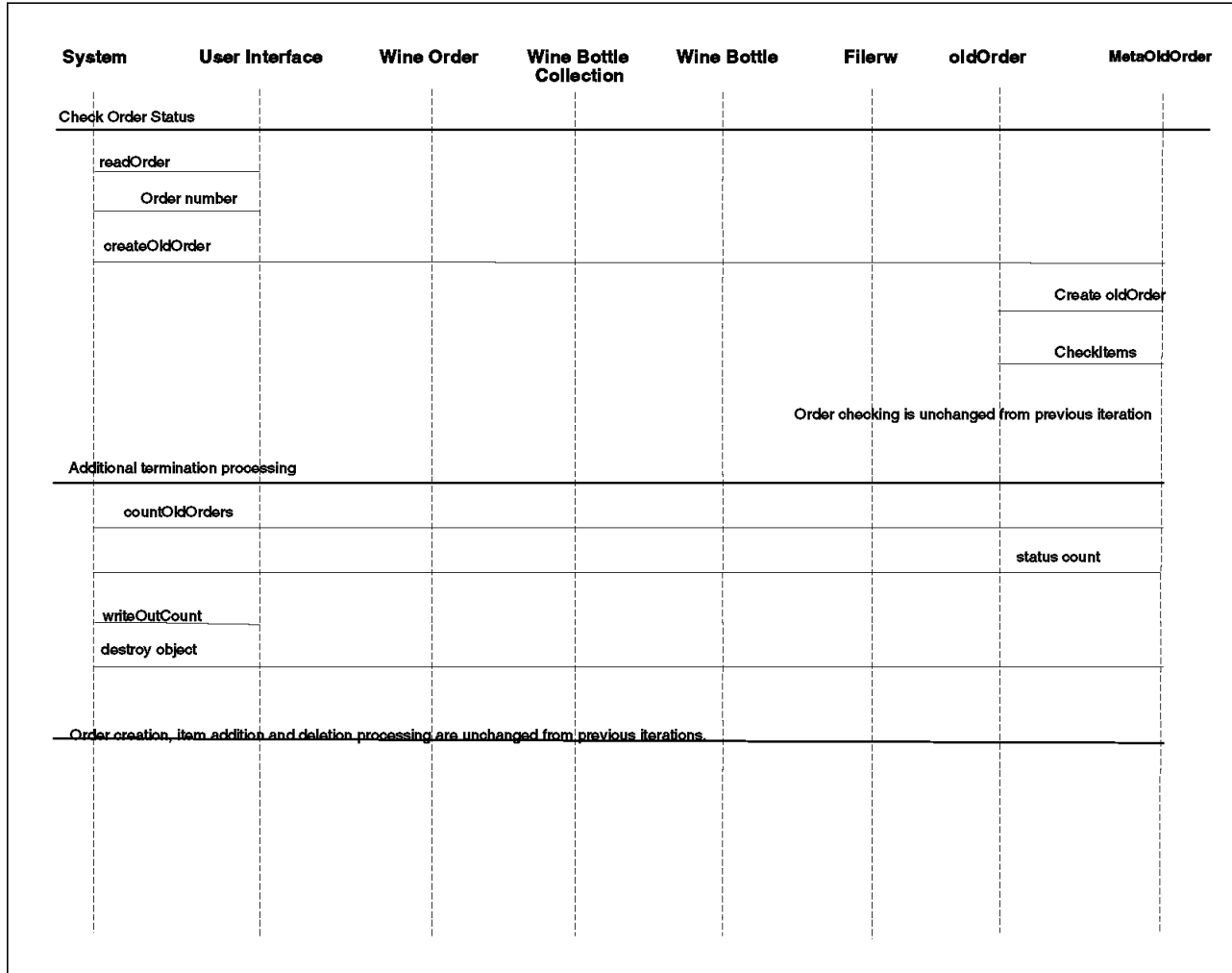


Figure 58. Object Interaction Diagram – Fourth Iteration

## 20.2 Code Enhancements

For the fourth phase, we implement the `MetaOldOrder` object and its methods of `CountOldOrder` and `CreateOldOrder`. We also modify the `OldOrder` object to use the `MetaOldOrder` class, while still inheriting from the `WineOrder` class. The `UserInterface` class is extended by the addition of the `WriteOutCount` method. Finally, the client program, `Wine`, is altered to incorporate the metaclass and associated methods.

We start by adding the new methods to our existing classes, then create the new class, and finally, extend our client program.

*Adding WriteOutCount to UserInterface:* This addition is easily implemented. It takes a parameter from the count of old orders checked that the metaclass tallies, and displays that count to the system user.

*Adding WriteLost to UserInterface:* This addition is trivial. It takes a parameter of the requested order number and displays the number along with an appropriate message to the system user.

*Adding Metaclass Changes to OldOrder.:* To accommodate the use of a metaclass, the OldOrder class must inherit from it. This requires no changes to the data or procedure division code, but a switch is added to indicate that the order could not be found.

*Coding MetaOldOrder:* MetaOldOrder can be thought of as a layer of code interjected between the client program, Wine, and the OldOrder class. Conceptually, it is a gatekeeper to the OldOrder class. It inherits from SOMClass, not SOMObject as other classes have done. somDefaultInit is overridden to initialize our count field. The method CreateOldOrder invokes the CheckItems method of OldOrder and increment the count field. The method CountOldOrders returns the count of old orders to the invoker.

*Changing the Wine Client Program:* The client program requires that we invoke the metaclass's CreateOldOrder method in lieu of OldOrder's CheckItems method when we want to check an order. On return from this invocation, we need to check a flag that indicates whether the order could not be found. If the order was not on the order file, the WriteLost method of UserInterface is invoked, and the rest of the code in the client is bypassed. The original action prompt is redisplayed to the system user. At the conclusion of the program, we invoke the metaclass's CountOldOrder method and pass the returned count field to the WriteOutCount method of the UserInterface.

---

## 20.3 Code Commentary

This section contains extracts of important features of the client and class programs with descriptions of how the code works.

### 20.3.1 Wine Client

The wine program is updated to flag any old checked orders. If they have, then the count of the OldOrders checked is obtained from the metaclass and passed to the UserInterface for display. When checking an OldOrder, the CreateOldOrder method of the metaclass, MetaOldOrder, is invoked instead of the CheckItems method of the OldOrder class, as was previously done.

To avoid redundancy, only the changed parts of the program are presented below.

```
process pgmname(longmixed) test
    IDENTIFICATION DIVISION.
    PROGRAM-ID. "Wine".
    .
    .
    .
    WORKING-STORAGE SECTION.
    .
    .
    .
```

```
01 metaObj          USAGE OBJECT REFERENCE METAClass OldOrder.
```

This code specifies the handle for the metaclass. It has an object reference to the class it has knowledge of.

```
.
.
.
01 STATUS-FLAG          PIC X.
   88 NO-STATUS-SELECTED      VALUE LOW-VALUE.
   88 STATUS-SELECTED        VALUE HIGH-VALUE.
.
.
.
```

This flag controls whether or not we invoke the classes and methods associated with displaying how many OldOrders were checked. It initializes to a low-value and sets to a high-value if we enter the CHECK-OLD-ORDER paragraph.

```
.
.
.
01 OUT-ORDERS          PIC S9(4)      COMP.
01 META-PARMS.
   05 univObj          USAGE OBJECT REFERENCE.
   05 LOST-FLAG        PIC X.
   05 OUT-ITEMS.
   05 OUT-COUNT        PIC S9(4).
       10 OUT-ITEM      OCCURS 1 TO 64 TIMES
                       DEPENDING ON OUT-COUNT
                       INDEXED BY OUT-INDEX.
               15 OUT-TYPE      PIC X(20).
               15 OUT-COST      PIC 999V99.
.
.
.
```

This code adds a field for the counter to indicate how many OldOrders were checked and includes a parameter structure for the metaclass to return to us. The metaclass returns a generic object reference, as well as the substructure of out-of-stock items which existed as a "01" level in the previous iteration.

```
PROCEDURE DIVISION.
  MOVE LOW-VALUES TO STATUS-FLAG.
.
.
.
```

This code initializes the control flag.

```
.
.
.
  PERFORM UNTIL PROCESS (1:4) = "EXIT"
    EVALUATE PROCESS (1:3)
      WHEN "STA"
        PERFORM CHECK-OLD-ORDER THRU CHECK-EXIT
        MOVE HIGH-VALUE TO STATUS-FLAG
    .
  .
.
```

```
END-PERFORM.
```

This code turns on the flag to indicate an OldOrder has been checked.

```
IF STATUS-SELECTED
  THEN PERFORM GET-COUNT THRU GET-EXIT.
.
.
.
```

This code checks the flag and performs a paragraph that obtains the number of OldOrders checked and displays it.

```
CHECK-OLD-ORDER.
  INVOKE userObj  "ReadOrder"    RETURNING  ORDER-NUMBER.
  INVOKE OldOrder "CreateOldOrder" USING  ORDER-NUMBER
                                RETURNING  META-PARMS.
.
.
.
```

This code invokes the CreateOldOrder method of the metaclass, which creates an OldOrder object. We no longer explicitly invoke somNew on the OldOrder object in the client program.

```
IF LOST-FLAG = HIGH-VALUE
  THEN INVOKE userObj "WriteLost" USING ORDER-NUMBER
  GO TO CHECK-EXIT.
```

Here we are merely checking a flag to see if the order was found. If it was not, we invoke a method in userinterface to write out an appropriate message and branch to the exit of the performed paragraph.

```
GET-COUNT.
  INVOKE univObj  "somGetClass"    RETURNING metaObj.
  INVOKE metaObj  "CountOldOrders" RETURNING OUT-ORDERS.
  INVOKE userObj  "WriteOutCount"  USING OUT-ORDERS.
  INVOKE metaObj  "somFree".
Get-EXIT.
EXIT.
```

In this code sample, we invoke the somGetClass method, inherited from SOMClass to obtain the handle of the metaclass object. Then, using this handle, we invoke the CountOldOrders method of the metaclass to obtain the number of OldOrders checked. This count is subsequently passed to the WriteOutCount method of the UserInterface class for display to the user. Finally, the metaObj is destroyed using somFree.

## 20.3.2 OldOrder Class

The changes required here are minimal as shown below.

```
IDENTIFICATION DIVISION.
CLASS-ID.  "OldOrder"  INHERITS WineOrder
                                METAClass MetaOldOrder.
```

We change the CLASS-ID statement to reflect the metaclass MetaOldOrder. METAClass is a keyword indicating the metaclass to be used.



```

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
    CLASS OldOrder          IS "OldOrder"
    CLASS MetaOldOrder      IS "MetaOldOrder"
.
.
.

```

This code adds MetaOldOrder to the REPOSITORY.

```

.
.
.
IF LS-ORDER-NUMBER NOT = WSO-ORDER-NUMBER
    THEN MOVE HIGH-VALUE TO LS-LOST-FLAG
    EXIT METHOD
ELSE
    MOVE LOW-VALUE TO LS-LOST-FLAG.
.
.
.

```

We also add the LOST-FLAG to the structure returned by OldOrder. After returning from the XReadOrder method, the input order is compared to the record returned. If they are not equal, the flag is set and the method exited; it is turned off if they are equal.

### 20.3.3 UserInterface Class

The UserInterface class has the new method WriteOutCount added to it.

```

IDENTIFICATION DIVISION.
METHOD-ID. "WriteOutCount".
DATA DIVISION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
01 LS-COUNT                  PIC S9(4)   COMP.

PROCEDURE DIVISION          USING      LS-OUT-ITEMS.
    DISPLAY LS-COUNT " orders checked".
    EXIT METHOD.
END METHOD "WriteOutCount".

```

This method accepts the count from the invoker and displays it with a meaningful (at least to the system user) literal.

```

IDENTIFICATION DIVISION.
METHOD-ID. "WriteLost".
DATA DIVISION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
01 LS-ORDER-NUMBER          PIC 9(5).

PROCEDURE DIVISION          USING      LS-ORDER-NUMBER.
    DISPLAY LS-ORDER-NUMBER "not on error file".
    EXIT METHOD.
END METHOD "WriteLost".

```

## 20.3.4 MetaOldOrder Class

The MetaOldOrder class is new for this iteration. It has three methods, one of which overrides somDefaultInit. CreateOldOrder creates an OldOrder object and invokes its CheckItems method and increments a counter. CountOldOrders returns the counter of OldOrders checked to the invoker.

```
process pgmname(mixed) test
  IDENTIFICATION DIVISION.
  CLASS-ID.    MetaOldOrder    INHERITS SOMClass.
  ENVIRONMENT DIVISION.
```

The CLASS-ID shows our metaclass inheriting from SOMClass instead of SOMObject, as most of our other classes have. This is needed to inherit metaclass methods from SOM.

```
  CONFIGURATION SECTION.
  REPOSITORY.
    CLASS MetaOldOrder      IS "MetaOldOrder"
    CLASS OldOrder          IS "OldOrder"
    CLASS SOMClass          IS "SOMClass".
```

Our REPOSITORY indicates that we use OldOrder, SOMClass, and MetaOldOrder.

```
  DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 STATUS-COUNT          PIC S9(4)      COMP.
```

The count field is coded in the WORKING-STORAGE SECTION. It initializes to zero and increments each time an OldOrder is checked.

```
  IDENTIFICATION DIVISION.
  METHOD-ID. "somDefaultInit"  OVERRIDE.
  PROCEDURE DIVISION.
    MOVE ZERO TO STATUS-COUNT.
    EXIT METHOD.
  END METHOD "somDefaultInit".
```

This code overrides somDefaultInit to initialize the count field.

```
  IDENTIFICATION DIVISION.
  METHOD-ID. "CreateOldOrder".
  DATA DIVISION.
  LINKAGE SECTION.
    01 LS-ORDER-NUMBER      PIC 9(5).
    01 LS-RETURN-PARMS.
      05 univObj            USAGE OBJECT REFERENCE.
      05 LS-CHECK-PARMS.
        10 LS-LOST-FLAG     PIC X.
        10 LS-OUT-ITEMS.
          15 LS-OUT-COUNT    PIC S9(4).
          15 LS-OUT-ITEM     OCCURS 1 TO 64
                               DEPENDING ON LS-OUT-COUNT
                               INDEXED BY  LS-INDEX.
        20 LS0-TYPE         PIC X(20).
        20 LS0-COST         PIC 999V99.
```

In the CreateOldOrder method, we use the order number as a parameter, and return a structure containing the out-of-stock items and a generic object reference to point to self, or this instance of MetaOldOrder.

```

PROCEDURE DIVISION
    USING      LS-ORDER-NUMBER
    RETURNING  LS-RETURN-PARMS.

    IF LS-ORDER-NUMBER > 0
        THEN INVOKE SELF "somNew"      RETURNING univObj
        INVOKE univObj "CheckItems"
        USING LS-ORDER-NUMBER
        RETURNING LS-CHECK-PARMS
        ADD 1 TO STATUS-COUNT
    END-IF.
    EXIT METHOD.
END METHOD "CreateOldOrder".

```

In the PROCEDURE DIVISION, the code creates a MetaOldOrder object by invoking somNew on self. Then, using the handle of the generic object reference, it invokes the CheckItems method, which is inherited from OldOrder. Upon return, it increments the count field.

```

IDENTIFICATION DIVISION.
METHOD-ID. "CountOldOrders".
DATA DIVISION.
LINKAGE SECTION.
01 LS-STATUS-COUNT          PIC S9(4)      COMP.
PROCEDURE DIVISION          RETURNING  LS-STATUS-COUNT.
    MOVE STATUS-COUNT TO LS-STATUS-COUNT.
    EXIT METHOD.
END METHOD "CountOldOrders".

```

In the CountOldOrders method, the code returns the count field to the invoker.

```
END CLASS MetaOldOrder.
```

The class definition for MetaOldOrder ends.

### 20.3.4.1 Conclusion

After the fourth iteration of our development effort, we have met all known functional requirements for the business process.

There are many other solutions to this rather simple problem. There are numerous variations on the class structure presented for this application. Nonetheless, the exercise illustrates the use of objects implemented in object-oriented COBOL as applied to a business process in an iterative development cycle using SOM.

As businesses grow and change and as additional knowledge is gained, business processes also change. Any system must accommodate these changes easily and in a robust manner. As business evolves, so must the system. This evolution is accommodated much more effectively with an object-oriented approach than was possible under previous development practices.



---

## Appendix A. Example One Source Code

This appendix lists all the source code modules for Example One.

---

### A.1 Example One – UserInterface Class Code

```
IDENTIFICATION DIVISION.
CLASS-ID. UserInterface Inherits SOMObject.
ENVIRONMENT DIVISION.
Configuration Section.
Repository.
    CLASS SOMObject IS "SOMObject"
    CLASS UserInterface IS "UserInt".
DATA DIVISION.
Working-Storage Section.
01  User-action      Pic X(10).
    88 User-add      Value "Addbott".
    88 User-delete   Value "Deletebott".
    88 User-end      Value "End".
01  User-Bottle      Pic X(20).
PROCEDURE DIVISION.
*
IDENTIFICATION DIVISION.
METHOD-ID. "ReadInput".
DATA DIVISION.
Linkage Section.
01  Action          Pic X(10).
01  Bottle          Pic X(20).
PROCEDURE DIVISION Using Bottle Action.
    Display "Enter the action : add, delete, end"
    Accept action from SYSIN
    Move Function Upper-case(action) to Action
    Evaluate action
        When "ADD"
            Set User-add to TRUE
            Perform Get-item
        When "DELETE"
            Set User-delete to TRUE
            Perform Get-item
        When "END"
            Set User-end to TRUE
    End-evaluate
    Move User-action to action
    Exit Method.

Get-item.
    Display "Enter the item"
    Accept Bottle from SYSIN
    Move Bottle to User-Bottle.

END METHOD "ReadInput".

IDENTIFICATION DIVISION.
METHOD-ID. "WriteMessage".
DATA DIVISION.
Working-Storage Section.
```

```

01  action      Pic X(10).
01  bottle      Pic X(20).
Linkage Section.
01  Flag        Pic 9.
PROCEDURE DIVISION Using Flag.
    Move user-Action to Action
    Move user-Bottle to Bottle
    IF flag = 0
        Display action " successfully completed on " bottle
    ELSE
        Display action " unsuccessfully completed on " bottle
    END-IF.
    Exit Method.
END METHOD "WriteMessage".
*
IDENTIFICATION DIVISION.
METHOD-ID. "Writeoutput".
DATA DIVISION.
Working-Storage Section.
77  Formatted-cost Pic $Z,ZZZ,ZZ9.99.
Linkage Section.
01  Total-cost      Pic 9(7)V99.
01  Case-number     Pic 9(5).
PROCEDURE DIVISION Using Total-cost Case-number.
    Move total-cost to Formatted-cost
    Display "Your order costs " Formatted-cost
    Display "Your case number is " Case-number
    Exit Method.
END METHOD "Writeoutput".
END CLASS UserInterface.

```

---

## A.2 Example One – WineCase Class Code

```

IDENTIFICATION DIVISION.
CLASS-ID. Winecase Inherits SOMObject.
ENVIRONMENT DIVISION.
Configuration Section.
Repository.
    CLASS SOMObject IS "SOMObject"
    CLASS Winecase IS "Winecase".
DATA DIVISION.
Working-Storage Section.
01  Case-Number      Pic 9(5).
01  Case-date        Pic X(8).
01  Case-Count       Pic 99.
01  Case-Contents.
    05  Case-Entry occurs 12 times.
        10  Case-Bottle Pic X(20).
PROCEDURE DIVISION.
*
*
*
IDENTIFICATION DIVISION.
METHOD-ID. "somDefaultInit" OVERRIDE.
PROCEDURE DIVISION.
    Compute Case-number = Function Random (99999)
    Move "01011996" to Case-date
    Move 0 to Case-count

```

```

        Initialize Case-Contents.
        Exit Method.
END METHOD "somDefaultInit".
IDENTIFICATION DIVISION.
METHOD-ID. "AddBott".
DATA DIVISION.
Working-Storage Section.
77  sub      Pic 99 VALUE 0.
01  Found-Flag      Pic 9.
    88  found        VALUE 0.
    88  not-found    VALUE 1.
Linkage Section.
01  In-bottle      Pic X(20).
01  Add-flag       Pic 9.
PROCEDURE DIVISION USING In-bottle Add-flag.
    Set not-found to True
    Move 1 to Add-flag
    Perform varying sub from 1 by 1
        until (sub > 12) or (found)
        IF Case-Bottle(sub) = SPACES
            Move in-bottle to Case-Bottle(sub)
            Add 1 to Case-Count
            Move 0 to Add-flag
            Set found to TRUE
        END-IF
    End-Perform.
    Exit method.
END METHOD "AddBott".

IDENTIFICATION DIVISION.
METHOD-ID. "RemoveBott".
DATA DIVISION.
Working-Storage Section.
77  sub      Pic 99 VALUE 0.
01  Found-Flag      Pic 9.
    88  found        VALUE 0.
    88  not-found    VALUE 1.
Linkage Section.
01  Out-bottle      Pic X(20).
01  Delete-flag     Pic 9.
PROCEDURE DIVISION USING Out-bottle Delete-flag.
    Set not-found to True
    Move 1 to Delete-flag
    Perform varying sub from 1 by 1
        until (sub > 12) or (found)
        IF Case-Bottle(sub) = Out-bottle
            Move SPACES to Case-Bottle(sub)
            Subtract 1 from Case-Count
            Move 0 to Delete-flag
            Set found to TRUE
        END-IF
    End-Perform.
    Exit method.
END METHOD "RemoveBott".

IDENTIFICATION DIVISION.
METHOD-ID. "CalculateCost".
DATA DIVISION.
Working-Storage Section.

```

```

77  sub      Pic 99 VALUE 0.
77  cost     Pic 9(5)V99.
Linkage Section.
01  Total-cost      Pic 9(7)V99.
PROCEDURE DIVISION using Total-cost.
    Move 0 to Total-cost
    Perform varying sub from 1 by 1
        until sub > case-count
        ADD 1    to Total-cost
    End-Perform.
    Exit method.
END METHOD "CalculateCost".
*
*
IDENTIFICATION DIVISION.
METHOD-ID. "GetCaseNumber".
DATA DIVISION.
Linkage Section.
01  Case-num      Pic 9(5).
PROCEDURE DIVISION using Case-num.
    Move Case-number to Case-num.
    Exit method.
END METHOD "GetCaseNumber".

IDENTIFICATION DIVISION.
METHOD-ID. "DescribeCase".
ENVIRONMENT DIVISION.
Input-Output Section.
File-Control.
    SELECT case-file ASSIGN to CaseData
    File Status is Data-key
    Organization is Line Sequential.
DATA DIVISION.
File Section.
FD  case-file External
    Record contains 255.
01  case-record  Pic X(255).
Working-Storage Section.
01  Data-key      Pic X(2).
01  print-line.
    05  print-case-number      Pic 9(5).
    05  print-case-date        Pic X(8).
    05  print-case-count       Pic 99.
    05  print-case-contents.
        10  print-case-entry  occurs 12 times.
        15  print-case-bottle  Pic X(20).
PROCEDURE DIVISION.
    Open Output case-file
    Move case-number  to print-case-number.
    Move case-date    to print-case-date.
    Move case-count   to print-case-count.
    Move case-contents to print-case-contents.
    Write case-record FROM print-line.
    Close case-file.
    Exit method.
END METHOD "DescribeCase".
END CLASS Winecase.

```



---

### A.3 Example One – Wine Client Class Code

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Wine.
ENVIRONMENT DIVISION.
Configuration Section.
Repository.
    CLASS SOMObject      IS "SOMObject"
    CLASS Case           IS "Winecase"
    CLASS UserInterface IS "UserInt".
DATA DIVISION.
Working-Storage Section.
77 caseObj              Usage Object Reference Case.
77 userObj              Usage Object Reference UserInterface.
77 Case-number          Pic 9(5).
77 total-cost           Pic 9(7)V99.
77 action               Pic X(10).
77 bottle               Pic X(20).
77 flag                 Pic X.
PROCEDURE DIVISION.
    Invoke UserInterface "somNew" RETURNING userObj
    Invoke Case          "somNew" RETURNING caseObj
    Invoke userObj "ReadInput" Using bottle action
    Perform until action = "End"
        IF action(1:3) = "Add"
            Invoke caseObj "AddBott" Using bottle flag
        ELSE
            Invoke caseObj "RemoveBott" Using bottle flag
        END-IF
        Invoke userObj "WriteMessage" Using flag
        Invoke userObj "ReadInput" Using bottle action
    End-Perform
    Invoke caseObj "CalculateCost" using total-cost
    Invoke caseObj "GetCaseNumber" Using case-number
    Invoke userObj "WriteOutput" Using total-cost case-number
    Invoke caseObj "DescribeCase"
    Invoke caseObj "somFree"
    Invoke userObj "somFree"
    STOP RUN.
END PROGRAM Wine.
```



---

## Appendix B. Example Two Source Code

This appendix lists all the source modules for Example Two.

---

### B.1 Example Two – UserInterface Class Code

```
IDENTIFICATION DIVISION.
CLASS-ID. "UserInt" Inherits SOMObject.
ENVIRONMENT DIVISION.
Configuration Section.
Repository.
    CLASS SOMObject IS "SOMObject"
    CLASS UserInterface IS "UserInt".
DATA DIVISION.
Working-Storage Section.
01  User-action      Pic X(10).
    88 User-add      Value "Addbott".
    88 User-delete   Value "Deletebott".
    88 User-end      Value "End".
01  User-Bottle      Pic X(20).
PROCEDURE DIVISION.
IDENTIFICATION DIVISION.
METHOD-ID. "ReadRequest".
DATA DIVISION.
Linkage Section.
01  Request          Pic X(6).
PROCEDURE DIVISION Using Request.
    Display "Enter the request: new, status"
    Accept request from SYSIN
    Move Function Upper-case(request) to Request
    Exit Method.
END METHOD "ReadRequest".
IDENTIFICATION DIVISION.
METHOD-ID. "ReadInput1".
DATA DIVISION.
Linkage Section.
01  Bottle           Pic X(20).
01  Action           Pic X(10).
PROCEDURE DIVISION Using Bottle Action.
    Display "Enter the action : add, delete, end"
    Accept action from SYSIN
    Move Function Upper-case(action) to Action
    Evaluate action
        When "ADD"
            Set User-add to TRUE
            Perform Get-item
        When "DELETE"
            Set User-delete to TRUE
            Perform Get-item
        When "END"
            Set User-end to TRUE
    End-evaluate
    Move User-action to action
    Exit Method.
Get-item.
    Display "Enter the item"
```

```

        Accept Bottle from SYSIN
        Move Bottle to User-Bottle.
    END METHOD "ReadInput1".
IDENTIFICATION DIVISION.
METHOD-ID. "ReadInput2".
DATA DIVISION.
Linkage Section.
01  Acct-num      Pic 9(5).
PROCEDURE DIVISION Using Acct-num.
    Display "Enter the account number."
    Accept Acct-num from SYSIN
    Exit Method.
END METHOD "ReadInput2".
IDENTIFICATION DIVISION.
METHOD-ID. "WriteMessage".
DATA DIVISION.
Working-Storage Section.
01  action        Pic X(10).
01  bottle        Pic X(20).
Linkage Section.
01  Flag          Pic 9.
PROCEDURE DIVISION Using Flag.
    Move user-Action to Action
    Move user-Bottle to Bottle
    IF flag = 0
        Display action " successfully completed on " bottle
    ELSE
        Display action " unsuccessfully completed on " bottle
    END-IF.
    Exit Method.
END METHOD "WriteMessage".
IDENTIFICATION DIVISION.
METHOD-ID. "Writeoutput".
DATA DIVISION.
Working-Storage Section.
77  Formatted-cost Pic $Z,ZZZ,ZZ9.99.
Linkage Section.
01  Total-cost     Pic 9(7)V99.
01  Case-number    Pic 9(5).
PROCEDURE DIVISION Using Total-cost Case-number.
    Move total-cost to Formatted-cost
    Display "Your order costs " Formatted-cost
    Display "Your case number is " Case-number
    Exit Method.
END METHOD "Writeoutput".
IDENTIFICATION DIVISION.
METHOD-ID. "WriteStatus".
DATA DIVISION.
Working-Storage Section.
77  sub            Pic 99 Value 99.
Linkage Section.
01  Out-table.
    05  Out-Entry occurs 12 times.
        10  Out-Bottle Pic X(20).
01  Out-count      Pic 99.
PROCEDURE DIVISION Using Out-table Out-count.
    IF out-count > 0
        Perform varying sub from 1 by 1
            until sub > out-count

```

```

        Display "Out of stock " Out-Bottle(sub)
    End-Perform
End-IF
Exit Method.
END METHOD "WriteStatus".
END CLASS "UserInt".

```

---

## B.2 Example Two – WineCase Class Code

```

IDENTIFICATION DIVISION.
CLASS-ID. "WineCase" Inherits SOMObject.
ENVIRONMENT DIVISION.
Configuration Section.
Repository.
    CLASS SOMObject IS "SOMObject"
    CLASS Winecase IS "WineCase".
DATA DIVISION.
Working-Storage Section.
01 Case-Number      Pic 9(5).
01 Case-date        Pic X(8).
01 Case-Count       Pic 99.
01 Case-Contents.
    05 Case-Entry occurs 12 times.
    10 Case-Bottle  Pic X(20).
PROCEDURE DIVISION.
*
*
*
IDENTIFICATION DIVISION.
METHOD-ID. "somDefaultInit" OVERRIDE.
PROCEDURE DIVISION.
    Compute Case-number = Function Random (99999)
    Move "01011996" to Case-date
    Move 0 to Case-count
    Initialize Case-Contents.
    Exit Method.
END METHOD "somDefaultInit".
IDENTIFICATION DIVISION.
METHOD-ID. "SetInstanceData".
DATA DIVISION.
Linkage Section.
01 In-case.
    05 in-case-number  Pic 9(5).
    05 in-case-date    Pic X(8).
    05 in-case-count   Pic 99.
    05 in-case-contents.
        10 in-case-entry occurs 12 times.
        15 in-case-bottle Pic X(20).
PROCEDURE DIVISION USING In-case.
    Move in-case-number to case-number
    Move in-case-date to case-date
    Move in-case-count to case-count
    Move in-case-contents to case-contents
    Exit method.
END METHOD "SetInstanceData".
IDENTIFICATION DIVISION.
METHOD-ID. "GetInstanceData".
DATA DIVISION.

```

```

Linkage Section.
01 out-case.
   05 out-case-number    Pic 9(5).
   05 out-case-date      Pic X(8).
   05 out-case-count     Pic 99.
   05 out-case-contents.
       10 out-case-entry occurs 12 times.
           15 out-case-bottle    Pic X(20).
PROCEDURE DIVISION Using out-case.
   Move case-number    to out-case-number
   Move case-date      to out-case-date
   Move case-count     to out-case-count
   Move case-contents to out-case-contents
   Exit method.
END METHOD "GetInstanceData".

```

```

IDENTIFICATION DIVISION.
METHOD-ID. "AddBott".
DATA DIVISION.
Working-Storage Section.
77 sub    Pic 99 VALUE 0.
01 Found-Flag    Pic 9.
   88 found      VALUE 0.
   88 not-found  VALUE 1.
Linkage Section.
01 In-bottle    Pic X(20).
01 Add-flag     Pic 9.
PROCEDURE DIVISION USING In-bottle Add-flag.
   Set not-found to True
   Move 1 to Add-flag
   Perform varying sub from 1 by 1
       until (sub > 12) or (found)
       IF Case-Bottle(sub) = SPACES
           Move in-bottle to Case-Bottle(sub)
           Add 1 to Case-Count
           Move 0 to Add-flag
           Set found to TRUE
       END-IF
   End-Perform.
   Exit method.
END METHOD "AddBott".

```

```

IDENTIFICATION DIVISION.
METHOD-ID. "RemoveBott".
DATA DIVISION.
Working-Storage Section.
77 sub    Pic 99 VALUE 0.
01 Found-Flag    Pic 9.
   88 found      VALUE 0.
   88 not-found  VALUE 1.
Linkage Section.
01 Out-bottle    Pic X(20).
01 Delete-flag   Pic 9.
PROCEDURE DIVISION USING Out-bottle Delete-flag.
   Set not-found to True
   Move 1 to Delete-flag
   Perform varying sub from 1 by 1
       until (sub > 12) or (found)

```

```

        IF Case-Bottle(sub) = Out-bottle
            Move SPACES to Case-Bottle(sub)
            Subtract 1 from Case-Count
            Move 0 to Delete-flag
            Set found to TRUE
        END-IF
    End-Perform.
    Exit method.
END METHOD "RemoveBott".

IDENTIFICATION DIVISION.
METHOD-ID. "CalculateCost".
DATA DIVISION.
Working-Storage Section.
77  sub    Pic 99 VALUE 0.
77  cost   Pic 9(5)V99.
Linkage Section.
01  Total-cost      Pic 9(7)V99.
PROCEDURE DIVISION using Total-cost.
    Move 0 to Total-cost
    Perform varying sub from 1 by 1
        until sub > case-count
        ADD 1      to Total-cost
    End-Perform.
    Exit method.
END METHOD "CalculateCost".
*
*
IDENTIFICATION DIVISION.
METHOD-ID. "GetCaseNumber".
DATA DIVISION.
Linkage Section.
01  Case-num      Pic 9(5).
PROCEDURE DIVISION using Case-num.
    Move Case-number to Case-num.
    Exit method.
END METHOD "GetCaseNumber".

IDENTIFICATION DIVISION.
METHOD-ID. "DescribeCase".
ENVIRONMENT DIVISION.
Input-Output Section.
File-Control.
    SELECT case-file ASSIGN to CaseData
    File Status is Data-key
    Organization is Line Sequential.
DATA DIVISION.
File Section.
FD  case-file External
    Record contains 255.
01  case-record  Pic X(255).
Working-Storage Section.
01  Data-key      Pic X(2).
01  print-line.
    05  print-case-number      Pic 9(5).
    05  print-case-date        Pic X(8).
    05  print-case-count       Pic 99.
    05  print-case-contents.
        10  print-case-entry  occurs 12 times.

```

```

15  print-case-bottle      Pic X(20).
PROCEDURE DIVISION.
  Open Output case-file
  Move case-number      to print-case-number.
  Move case-date        to print-case-date.
  Move case-count       to print-case-count.
  Move case-contents    to print-case-contents.
  Write case-record FROM print-line.
  Close case-file.
  Exit method.
END METHOD "DescribeCase".
END CLASS "WineCase".

```

---

### B.3 Example Two – Wine Client Class Code

```

IDENTIFICATION DIVISION.
PROGRAM-ID. "Wine".
ENVIRONMENT DIVISION.
Configuration Section.
Repository.
  CLASS SOMObject      IS "SOMObject"
  CLASS NewCase        IS "NewCase"
  CLASS OldCase        IS "OldCase"
  CLASS UserInt IS "UserInt".
DATA DIVISION.
Working-Storage Section.
77  univObj            Usage Object Reference.
77  userObj            Usage Object Reference UserInt.
77  Case-number        Pic 9(5).
77  total-cost         Pic 9(7)V99.
77  out-count          Pic 9(2).
77  request            Pic X(6).
77  bottle             Pic X(20).
77  action             Pic X(10).
77  flag              Pic X.
01  Case-Contents.
    05 Case-Entry occurs 12 times.
      10 Case-Bottle Pic X(20).
PROCEDURE DIVISION.
  Invoke UserInt "somNew" Returning userObj
  Invoke userObj "ReadRequest" Using request.
  IF request = "STATUS"
    Perform CheckOldCase
  ELSE
    Perform CreateNewCase
  END-IF.
  Invoke userObj "somFree"
  STOP RUN.

CheckOldCase.
  Invoke OldCase "somNew" Returning univObj
  Invoke userObj "ReadInput2" Using Case-number
  Invoke univObj "ReadCase" Using Case-number flag
  Invoke univObj "CheckBott" Using Case-contents out-count
  Invoke userObj "WriteStatus" Using Case-contents out-count
  Invoke univObj "somFree".

CreateNewCase.

```



```

Invoke NewCase "somNew" Returning univObj
Invoke userObj "ReadInput1" Using bottle action
Perform until action = "End"
  IF action(1:3) = "Add"
    Invoke univObj "AddBott" Using bottle flag
  ELSE
    Invoke univObj "DeleteBott" Using bottle flag
  END-IF
Invoke userObj "WriteMessage" Using flag
Invoke userObj "ReadInput1" Using bottle action
End-Perform

Invoke univObj "CalculateCost" Using total-cost
Invoke univObj "GetCaseNumber" Using case-number
Invoke userObj "WriteOutput" Using total-cost case-number
Invoke univObj "DescribeCase"
Invoke univObj "somFree".

END PROGRAM "Wine".

```

---

## B.4 Example Two – NewCase Class Code

```

IDENTIFICATION DIVISION.
CLASS-ID. "NewCase" Inherits Winecase.
ENVIRONMENT DIVISION.
Configuration Section.
Repository.
  CLASS NewCase IS "NewCase"
  CLASS Winecase IS "WineCase".
DATA DIVISION.
PROCEDURE DIVISION.
*
*
*
END CLASS "NewCase".

```

---

## B.5 Example Two – OldCase Class Code

```

IDENTIFICATION DIVISION.
CLASS-ID. "OldCase" Inherits Winecase.
ENVIRONMENT DIVISION.
Configuration Section.
Repository.
  CLASS OldCase IS "OldCase"
  CLASS Winecase IS "WineCase".
DATA DIVISION.
PROCEDURE DIVISION.
*
*
*
IDENTIFICATION DIVISION.
METHOD-ID. "ReadCase".
ENVIRONMENT DIVISION.
Input-Output Section.
File-Control.
  SELECT the-case-file ASSIGN thecase
  Organization is Line Sequential.

```

```

DATA DIVISION.
File Section.
FD The-case-file External.
01 The-case-record    Pic X(255).
Working-Storage Section.
01  the-case.
    05  the-case-number    Pic 9(5).
    05  the-case-date      Pic X(8).
    05  the-case-count     Pic 99.
    05  the-case-contents.
        10  the-case-entry occurs 12 times.
            15  the-case-bottle    Pic X(20).
77  eof-flag    Pic 9.
88  eof        Value 0.
Linkage Section.
01  case-number    Pic 9(5).
PROCEDURE DIVISION Using Case-number.
    Open Input The-case-file
    Move 1 to eof-flag
    Perform until eof
        Read the-case-file into the-case
        At end
            Set eof to TRUE
        Not at end
            IF Case-number = the-case-number
                Invoke SELF "SetInstanceData" Using the-case
            END-IF
    End-Read
    End-Perform
    Close The-Case-file
    Exit Method.
END METHOD "ReadCase".
IDENTIFICATION DIVISION.
METHOD-ID. "Checkbott".
DATA DIVISION.
Working-Storage Section.
77  Random-setting    Pic 9(8) Usage Comp.
77  sub                Pic 99.
77  status-flag    Pic 9.
88  in-stock        VALUE 0.
88  out-stock       VALUE 1.
01  the-case.
    05  the-case-number    Pic 9(5).
    05  the-case-date      Pic X(8).
    05  the-case-count     Pic 99.
    05  the-case-contents.
        10  the-case-entry occurs 12 times.
            15  the-case-bottle    Pic X(20).
Linkage Section.
01  out-contents.
    05  out-entry occurs 12 times.
        10  out-bottle    Pic X(20).
01  out-count    Pic 99.
PROCEDURE DIVISION Using out-contents out-count.
    Invoke SELF "GetInstanceData" Using the-case
    Move 0 to out-count
    Perform varying sub from 1 by 1
        until sub > the-case-count
    Compute Random-setting = 0.5 + Function Random

```

```

Compute Random-setting = Function Integer (Random-setting)
IF Random-Setting = 1
    Set out-stock to TRUE
ELSE
    Set in-stock to TRUE
END-IF
IF out-stock
Add 1 to out-count
Move The-case-bottle(sub) to out-bottle(out-count)
END-IF
End-Perform
Exit method.
END METHOD "Checkbott".
*
END CLASS "OldCase".

```



---

## Appendix C. Example Three Source Code

This appendix lists all the source modules for Example Three.

---

### C.1 Example Three – UserInterface Call Code

```
IDENTIFICATION DIVISION.
CLASS-ID. "UserInt" Inherits SOMObject.
ENVIRONMENT DIVISION.
Configuration Section.
Repository.
    CLASS SOMObject IS "SOMObject"
    CLASS UserInterface IS "UserInt".
DATA DIVISION.
Working-Storage Section.
01  User-action      Pic X(10).
    88  User-add      Value "Addbott".
    88  User-delete   Value "Deletebott".
    88  User-end      Value "End".
01  User-Bottle      Pic X(20).
PROCEDURE DIVISION.
IDENTIFICATION DIVISION.
METHOD-ID. "ReadRequest".
DATA DIVISION.
Linkage Section.
01  Request          Pic X(6).
PROCEDURE DIVISION Using Request.
    Display "Enter the request: new, status"
    Accept request from SYSIN
    Move Function Upper-case(request) to Request
    Exit Method.
END METHOD "ReadRequest".
IDENTIFICATION DIVISION.
METHOD-ID. "ReadInput1".
DATA DIVISION.
Linkage Section.
01  Bottle           Pic X(20).
01  Action           Pic X(10).
PROCEDURE DIVISION Using Bottle Action.
    Display "Enter the action : add, delete, end"
    Accept action from SYSIN
    Move Function Upper-case(action) to Action
    Evaluate action
        When "ADD"
            Set User-add to TRUE
            Perform Get-item
        When "DELETE"
            Set User-delete to TRUE
            Perform Get-item
        When "END"
            Set User-end to TRUE
    End-evaluate
    Move User-action to action
    Exit Method.
Get-item.
    Display "Enter the item"
```

```

        Accept Bottle from SYSIN
        Move Bottle to User-Bottle.
    END METHOD "ReadInput1".
IDENTIFICATION DIVISION.
METHOD-ID. "ReadInput2".
DATA DIVISION.
Linkage Section.
01  Acct-num    Pic 9(5).
PROCEDURE DIVISION Using Acct-num.
    Display "Enter the account number."
    Accept Acct-num from SYSIN
    Exit Method.
END METHOD "ReadInput2".
IDENTIFICATION DIVISION.
METHOD-ID. "WriteMessage".
DATA DIVISION.
Working-Storage Section.
01  action      Pic X(10).
01  bottle      Pic X(20).
Linkage Section.
01  Flag        Pic 9.
PROCEDURE DIVISION Using Flag.
    Move user-Action to Action
    Move user-Bottle to Bottle
    IF flag = 0
        Display action " successfully completed on " bottle
    ELSE
        Display action " unsuccessfully completed on " bottle
    END-IF.
    Exit Method.
END METHOD "WriteMessage".
IDENTIFICATION DIVISION.
METHOD-ID. "Writeoutput".
DATA DIVISION.
Working-Storage Section.
77  Formatted-cost  Pic $Z,ZZZ,ZZ9.99.
Linkage Section.
01  Total-cost      Pic 9(7)V99.
01  Case-number     Pic 9(5).
PROCEDURE DIVISION Using Total-cost Case-number.
    Move total-cost to Formatted-cost
    Display "Your order costs " Formatted-cost
    Display "Your case number is " Case-number
    Exit Method.
END METHOD "Writeoutput".
IDENTIFICATION DIVISION.
METHOD-ID. "WriteStatus".
DATA DIVISION.
Working-Storage Section.
77  sub            Pic 99 Value 99.
Linkage Section.
01  Out-table.
    05  Out-Entry occurs 12 times.
        10  Out-Bottle  Pic X(20).
01  Out-count      Pic 99.
PROCEDURE DIVISION Using Out-table Out-count.
    IF out-count > 0
        Perform varying sub from 1 by 1
            until sub > out-count

```

```

        Display "Out of stock " Out-Bottle(sub)
    End-Perform
End-IF
Exit Method.
END METHOD "WriteStatus".
END CLASS "UserInt".

```

---

## C.2 Example Three – WineCase Class Code

```

IDENTIFICATION DIVISION.
CLASS-ID. "WineCase" Inherits SOMObject.
ENVIRONMENT DIVISION.
Configuration Section.
Repository.
    CLASS SOMObject IS "SOMObject"
    CLASS Winecase IS "WineCase".
DATA DIVISION.
Working-Storage Section.
01 Case-Number      Pic 9(5).
01 Case-date        Pic X(8).
01 Case-Count       Pic 99.
01 Case-Contents.
    05 Case-Entry occurs 12 times.
    10 Case-Bottle  Pic X(20).
PROCEDURE DIVISION.
*
*
*
IDENTIFICATION DIVISION.
METHOD-ID. "somDefaultInit" OVERRIDE.
PROCEDURE DIVISION.
    Compute Case-number = Function Random (99999)
    Move "01011996" to Case-date
    Move 0 to Case-count
    Initialize Case-Contents.
    Exit Method.
END METHOD "somDefaultInit".
IDENTIFICATION DIVISION.
METHOD-ID. "SetInstanceData".
DATA DIVISION.
Linkage Section.
01 In-case.
    05 in-case-number  Pic 9(5).
    05 in-case-date    Pic X(8).
    05 in-case-count   Pic 99.
    05 in-case-contents.
        10 in-case-entry occurs 12 times.
        15 in-case-bottle Pic X(20).
PROCEDURE DIVISION USING In-case.
    Move in-case-number to case-number
    Move in-case-date to case-date
    Move in-case-count to case-count
    Move in-case-contents to case-contents
    Exit method.
END METHOD "SetInstanceData".
IDENTIFICATION DIVISION.
METHOD-ID. "GetInstanceData".
DATA DIVISION.

```

```

Linkage Section.
01 out-case.
   05 out-case-number    Pic 9(5).
   05 out-case-date      Pic X(8).
   05 out-case-count     Pic 99.
   05 out-case-contents.
       10 out-case-entry occurs 12 times.
           15 out-case-bottle    Pic X(20).
PROCEDURE DIVISION Using out-case.
   Move case-number    to out-case-number
   Move case-date      to out-case-date
   Move case-count     to out-case-count
   Move case-contents to out-case-contents
   Exit method.
END METHOD "GetInstanceData".

```

```

IDENTIFICATION DIVISION.
METHOD-ID. "AddBott".
DATA DIVISION.
Working-Storage Section.
77 sub    Pic 99 VALUE 0.
01 Found-Flag    Pic 9.
   88 found      VALUE 0.
   88 not-found  VALUE 1.
Linkage Section.
01 In-bottle    Pic X(20).
01 Add-flag     Pic 9.
PROCEDURE DIVISION USING In-bottle Add-flag.
   Set not-found to True
   Move 1 to Add-flag
   Perform varying sub from 1 by 1
       until (sub > 12) or (found)
       IF Case-Bottle(sub) = SPACES
           Move in-bottle to Case-Bottle(sub)
           Add 1 to Case-Count
           Move 0 to Add-flag
           Set found to TRUE
       END-IF
   End-Perform.
   Exit method.
END METHOD "AddBott".

```

```

IDENTIFICATION DIVISION.
METHOD-ID. "RemoveBott".
DATA DIVISION.
Working-Storage Section.
77 sub    Pic 99 VALUE 0.
01 Found-Flag    Pic 9.
   88 found      VALUE 0.
   88 not-found  VALUE 1.
Linkage Section.
01 Out-bottle    Pic X(20).
01 Delete-flag   Pic 9.
PROCEDURE DIVISION USING Out-bottle Delete-flag.
   Set not-found to True
   Move 1 to Delete-flag
   Perform varying sub from 1 by 1
       until (sub > 12) or (found)

```



```

        IF Case-Bottle(sub) = Out-bottle
            Move SPACES to Case-Bottle(sub)
            Subtract 1 from Case-Count
            Move 0 to Delete-flag
            Set found to TRUE
        END-IF
    End-Perform.
    Exit method.
END METHOD "RemoveBott".

IDENTIFICATION DIVISION.
METHOD-ID. "CalculateCost".
DATA DIVISION.
Working-Storage Section.
77 sub    Pic 99 VALUE 0.
77 cost   Pic 9(5)V99.
Linkage Section.
01 Total-cost      Pic 9(7)V99.
PROCEDURE DIVISION using Total-cost.
    Move 0 to Total-cost
    Perform varying sub from 1 by 1
        until sub > case-count
            ADD 1    to Total-cost
    End-Perform.
    Exit method.
END METHOD "CalculateCost".
*
*
IDENTIFICATION DIVISION.
METHOD-ID. "GetCaseNumber".
DATA DIVISION.
Linkage Section.
01 Case-num      Pic 9(5).
PROCEDURE DIVISION using Case-num.
    Move Case-number to Case-num.
    Exit method.
END METHOD "GetCaseNumber".

IDENTIFICATION DIVISION.
METHOD-ID. "DescribeCase".
ENVIRONMENT DIVISION.
Input-Output Section.
File-Control.
    SELECT case-file ASSIGN to CaseData
    File Status is Data-key
    Organization is Line Sequential.
DATA DIVISION.
File Section.
FD case-file External
    Record contains 255.
01 case-record Pic X(255).
Working-Storage Section.
01 Data-key      Pic X(2).
01 print-line.
    05 print-case-number      Pic 9(5).
    05 print-case-date        Pic X(8).
    05 print-case-count       Pic 99.
    05 print-case-contents.
    10 print-case-entry occurs 12 times.

```

```

15  print-case-bottle      Pic X(20).
PROCEDURE DIVISION.
  Open Output case-file
  Move case-number      to print-case-number.
  Move case-date        to print-case-date.
  Move case-count       to print-case-count.
  Move case-contents to print-case-contents.
  Write case-record FROM print-line.
  Close case-file.
  Exit method.
END METHOD "DescribeCase".
END CLASS "WineCase".

```

---

### C.3 Example Three – Wine Client Class Code

```

IDENTIFICATION DIVISION.
PROGRAM-ID. "Wine".
ENVIRONMENT DIVISION.
Configuration Section.
Repository.
  CLASS SOMObject      IS "SOMObject"
  CLASS NewCase        IS "NewCase"
  CLASS OldCase        IS "OldCase"
  CLASS UserInterface IS "UserInt".
DATA DIVISION.
Working-Storage Section.
77  univObj            Usage Object Reference.
77  metaObj            Usage Object Reference MetaClass OldCase.
77  userObj            Usage Object Reference UserInterface.
77  Case-number        Pic 9(5).
77  total-cost         Pic 9(7)V99.
77  out-count          Pic 9(2).
77  request            Pic X(6).
77  action             Pic X(10).
77  bottle             Pic X(20).
77  flag               Pic X.
01  Case-Contents.
    05 Case-Entry occurs 12 times.
      10 Case-Bottle   Pic X(20).
PROCEDURE DIVISION.
  Invoke UserInterface "somNew" Returning userObj
  Invoke userobj "ReadRequest" Using request.
  IF request = "STATUS"
  Perform CheckOldCase
  ELSE
  Perform CreateNewCase
  END-IF.
  Invoke userobj "somFree"
  STOP RUN.

CheckOldCase.
  Invoke userobj "ReadInput2" Using Case-number
  Invoke OldCase "CreateOldCase" Using Case-number univObj.
  Perform Until Case-number < 0
  Invoke univObj "CheckBott" Using Case-Contents out-count
  Invoke userObj "WriteStatus" Using Case-Contents out-count
  Invoke userobj "ReadInput2" Using Case-number
  Invoke OldCase "CreateOldCase" Using Case-number univObj

```

```

End-Perform

    Invoke univobj "somGetClass" Using metaObj
    Invoke metaobj "CountOldCase" Using out-count
    Invoke userobj "WriteMessage" Using out-count OMITTED
    Invoke metaObj "somFree".
CreateNewCase.
    Invoke NewCase "somNew" returning univobj
    Invoke userobj "ReadInput1" Using bottle action
    Perform until action = "End"
    IF action(1:3) = "Add"
        Invoke univObj "AddBott" Using bottle flag
    ELSE
        Invoke univObj "RemoveBott" Using bottle flag
    END-IF
    Invoke userObj "WriteMessage" Using flag
    Invoke userObj "ReadInput1" Using bottle action
End-Perform

    Invoke univObj "CalculateCost" Using total-cost
    Invoke univObj "GetCaseNumber" Using case-number
    Invoke userObj "WriteOutput" Using total-cost case-number
    Invoke univObj "DescribeCase"
    Invoke univObj "somFree".

END PROGRAM "Wine".

```

---

## C.4 Example Three – MetaOldCase Metaclass Code

```

IDENTIFICATION DIVISION.
CLASS-ID. "MetaOldCase" Inherits SOMClass.
ENVIRONMENT DIVISION.
Configuration Section.
Repository.
    CLASS MetaOldCase IS "MetaOldCase"
    CLASS OldCase IS "OldCase"
    CLASS SOMClass IS "SOMClass".
DATA DIVISION.
Working-Storage Section.
01 status-count Pic 99.
PROCEDURE DIVISION.
*
*
*
IDENTIFICATION DIVISION.
METHOD-ID. "somDefaultInit" OVERRIDE.

PROCEDURE DIVISION.
    Move 0 to status-count.
    Exit Method.
END METHOD "somDefaultInit".
IDENTIFICATION DIVISION.
METHOD-ID. "CreateOldCase".
DATA DIVISION.
Linkage Section.
01 Case-number Pic 9(5).
01 anObj USAGE OBJECT REFERENCE.

```

```

PROCEDURE DIVISION Using Case-number anObj.
    IF Case-number > 0
        Invoke SELF "somNew" Using anObj
        Invoke anObj "ReadCase" Using Case-number
        Add 1 to status-count
    END-IF
    Exit method.
END METHOD "CreateOldCase".
IDENTIFICATION DIVISION.
METHOD-ID. "CountOldCase".
DATA DIVISION.
Linkage Section.
01  out-count    Pic 9(2).
PROCEDURE DIVISION Using out-count.
    Move status-count to out-count.
    Exit method.
END METHOD "CountOldCase".
END CLASS "MetaOldCase".

```

---

## C.5 Example Three – OldCase Class Code

```

IDENTIFICATION DIVISION.
CLASS-ID. "OldCase" Inherits WineCase MetaClass MetaOldCase.
ENVIRONMENT DIVISION.
Configuration Section.
Repository.
    CLASS MetaOldCase IS "MetaOldCase"
    CLASS OldCase IS "OldCase"
    CLASS WineCase IS "WineCase".
DATA DIVISION.
PROCEDURE DIVISION.
*
*
*
IDENTIFICATION DIVISION.
METHOD-ID. "ReadCase".
ENVIRONMENT DIVISION.
Input-Output Section.
File-Control.
    SELECT the-case-file ASSIGN thecase
        Organization is Line Sequential.
DATA DIVISION.
File Section.
FD The-case-file External.
01 The-case-record    Pic X(255).
Working-Storage Section.
01  the-case.
    05  the-case-number    Pic 9(5).
    05  the-case-date      Pic X(8).
    05  the-case-count     Pic 99.
    05  the-case-contents.
        10 the-case-entry occurs 12 times.
        15 the-case-bottle Pic X(20).
77 eof-flag    Pic 9.
88 eof        Value 0.
Linkage Section.
01  case-number Pic 9(5).

```

```

PROCEDURE DIVISION Using Case-number.
  Open Input The-case-file
  Move 1 to eof-flag
  Perform until eof
    Read the-case-file into the-case
    At end
      Set eof to TRUE
    Not at end
      IF Case-number = the-case-number
        Invoke SELF "SetInstanceData" Using The-case
      END-IF
    End-Read
  End-Perform
  Close The-Case-file
  Exit Method.
END METHOD "ReadCase".
IDENTIFICATION DIVISION.
METHOD-ID. "CheckBott".
DATA DIVISION.
Working-Storage Section.
77 Random-setting Pic 9(8) Usage Comp.
77 sub Pic 99.
77 status-flag Pic 9.
88 in-stock VALUE 0.
88 out-stock VALUE 1.
01 the-case.
05 the-case-number Pic 9(5).
05 the-case-date Pic X(8).
05 the-case-count Pic 99.
05 the-case-contents.
10 the-case-entry occurs 12 times.
15 the-case-bottle Pic X(20).
Linkage Section.
01 out-contents.
05 out-entry occurs 12 times.
10 out-bottle Pic X(20).
01 out-count Pic 99.
PROCEDURE DIVISION Using out-contents out-count.
  Invoke SELF "GetInstanceData" Using The-case
  Move 0 to out-count
  Perform varying sub from 1 by 1
    until sub > the-case-count
  Compute Random-setting = 0.5 + Function Random
  Compute Random-setting = Function Integer (Random-setting)
  Display "Random number is " Random-setting
  IF Random-Setting = 1
    Set out-stock to TRUE
  ELSE
    Set in-stock to TRUE
  END-IF
  IF out-stock
    Add 1 to out-count
    Move The-case-bottle(sub) to out-bottle(out-count)
  END-IF
  End-Perform
  Exit method.
END METHOD "CheckBott".
END CLASS "OldCase".

```

---

## C.6 Example Three – NewCase Class Code

```
IDENTIFICATION DIVISION.  
CLASS-ID. "NewCase" Inherits Winecase.  
ENVIRONMENT DIVISION.  
Configuration Section.  
Repository.  
    CLASS NewCase IS "NewCase"  
    CLASS Winecase IS "WineCase".  
DATA DIVISION.  
PROCEDURE DIVISION.  
*  
*  
*  
END CLASS "NewCase".
```

---

## Appendix D. Wine Store Scenario – Iteration One Code

This appendix lists all the source modules for the first iteration of the Wine Store Scenario.

---

### D.1 Wine Client Code

```
process test pgmname(longmixed)
  IDENTIFICATION DIVISION.
  PROGRAM-ID. "Wine".

  ENVIRONMENT DIVISION.
  Configuration Section.
  Repository.
    CLASS SOMObject      IS "SOMObject"
    CLASS TheOrder       IS "WineOrder"
    CLASS UserInterface IS "UserInterface".

  DATA DIVISION.
  Working-Storage Section.
    01 orderObj          Usage Object Reference TheOrder.
    01 userObj           Usage Object Reference UserInterface.
    01 Item-Count        Pic 9(4).
    01 Max-items         Pic 9(4) Value 64.
    01 Order-number      Pic 9(5).
    01 Order-date        Pic X(8) Value "00/00/00".
    77 Item-type         Pic X(20).
    77 Item-cost         Pic 999V99.
    77 total-cost        Pic 9(7)V99.
    77 action            Pic X(10).
    77 Today             Pic X(21).
    77 flag              Pic X(4).
    01 WS-items.
      05 WS-count        Pic 9(4).
      05 WS-item         Occurs 1 to 64 times
                        Depending on WS-count
                        Indexed by WS-Index.
      10 WS-type         Pic X(20).
      10 WS-cost         Pic 999V99.

  PROCEDURE DIVISION.
    Invoke UserInterface "somNew" RETURNING userObj
    Invoke TheOrder      "somNew" RETURNING orderObj

    Move Function Current-date to Today
    Move Today(3:2) to Order-date(1:2)
    Move Today(5:2) to Order-date(4:2)
    Move Today(7:2) to Order-date(7:2)
    Compute Order-Number
      = Function Integer(10000 * Function Random )

    Invoke orderObj "InitOrder" Using Order-Number Order-Date

    Move Zero to Item-Count

    Invoke userobj "ReadAction" Returning action
```

```

Perform until action = "End"
      or Item-Count >= Max-Items
  IF action(1:3) = "Add"
    Add 1 to Item-Count
    Invoke userObj  "Readtype" Returning Item-type
    Invoke userObj  "ReadCost" Returning Item-cost
    Invoke orderObj "AddBott" Using Item-type Item-cost
                          Returning flag
    Invoke userObj  "WriteMessage" Using flag
  ELSE
    Invoke userObj "Readtype" Returning Item-type
    Invoke userObj "Readcost" Returning Item-cost
    Invoke orderObj "DeleteBott"
                          Using Item-type Item-cost Returning Flag
    Invoke userObj "WriteMessage" Using flag
    Subtract 1 from Item-count
  END-IF

  Invoke userObj "ReadAction" Returning action

End-Perform

Invoke orderObj "CalculateCost"    Returning total-cost
Invoke orderObj "GetOrderNumber"   Returning Order-number
Invoke orderObj "GetOrderDate"     Returning Order-date
Invoke userObj  "WriteOutput"
      Using total-cost Order-number Order-Date
Invoke orderObj "DescribeOrder"    Returning WS-items
Invoke userObj  "Writebottle"      Using WS-items
Invoke orderObj "ScrapOrder"
Invoke orderObj "somFree"
Invoke userObj  "somFree"
STOP RUN.
END PROGRAM "Wine".

```

---

## D.2 Wine Order Class Code

```

process test pgmname(longmixed)
  IDENTIFICATION DIVISION.
  CLASS-ID. "WineOrder" Inherits SOMObject.

  ENVIRONMENT DIVISION.
  Configuration Section.
  Repository.
    CLASS SOMObject IS "SOMObject"
    CLASS Bottle    IS "Bottle"
    CLASS WineOrder IS "WineOrder".

  DATA DIVISION.
  Working-Storage Section.
  01 Order-Number      Pic X(5).
  01 Order-date        Pic X(8).
  01 bottles.
    05 Bottle-count    Pic 9(4).
    05 Bottle-item     Occurs 1 to 64 times
                        Depending on Bottle-count
                        Indexed by Item-Count.

```



10 bottleObj                    Usage Object Reference Bottle.

PROCEDURE DIVISION.

IDENTIFICATION DIVISION.

METHOD-ID. "DescribeOrder".

DATA DIVISION.

Working-Storage Section.

01 WS-Type                    Pic x(20).

01 WS-Cost                   Pic 999V99.

Linkage Section.

01 LS-Items.

05 LS-Item-Count            Pic 9(4).

05 LS-Item                   Occurs 1 to 64 times  
                             Depending on LS-Item-Count  
                             Indexed by LS-Index.

10 LS-Type                   Pic x(20).

10 LS-Cost                   Pic 999V99.

PROCEDURE DIVISION Returning LS-items.

Move Bottle-count to LS-Item-Count

Set LS-Index to 1

Perform varying Item-Count from 1 by 1  
until (Item-Count > Bottle-count)

Invoke bottleObj(Item-Count) "GetCost"  
Returning WS-Cost

Move WS-Cost to LS-Cost (LS-Index)

Invoke bottleObj(Item-Count) "GetType"  
Returning WS-Type

Move WS-Type to LS-Type (LS-Index)

Set LS-Index up by 1

End-Perform

Exit method.

END METHOD "DescribeOrder".

IDENTIFICATION DIVISION.

METHOD-ID. "AddBott".

DATA DIVISION.

Working-Storage Section.

01 Found-Flag                Pic X(4).

88 found                    VALUE "0".

88 not-found                VALUE "1".

Linkage Section.

01 LS-Type                   Pic X(20).

01 LS-Cost                   Pic 999V99.

01 LS-flag                   Pic X(4).

PROCEDURE DIVISION USING LS-Type LS-Cost

Returning LS-flag.

Move "1" to LS-flag

Found-flag

Perform varying Item-Count from 1 by 1  
until (Item-Count > 64) or (found)

IF BottleObj(Item-Count) = NULL

Invoke Bottle "somNew"

Returning BottleObj(Item-Count)

Invoke bottleObj(Item-Count) "SetType"

Using LS-Type

Invoke bottleObj(Item-Count) "SetCost"

Using LS-Cost

Add 1 to Bottle-Count

```

        Move "0" to LS-flag
        Found-Flag
    END-IF
    End-Perform.
    Exit method.
END METHOD "AddBott".

IDENTIFICATION DIVISION.
METHOD-ID. "DeleteBott".
DATA DIVISION.
Working-Storage Section.
01  Found-Flag      Pic X(4).
    88  found        VALUE "0".
    88  not-found    VALUE "1".
Local-Storage Section.
77  Bott-Type       Pic X(20).
77  Bott-Cost        Pic 999V99.
Linkage Section.
01  LS-Type          Pic X(20).
01  LS-Cost           Pic 999V99.
01  Delete-flag      Pic x(4).
PROCEDURE DIVISION USING LS-Type LS-Cost
    Returning Delete-flag.
    Move "1" to Found-Flag
    Delete-flag
    Perform varying Item-Count from 1 by 1
        until (Item-Count > bottle-count) or (found)
        Invoke BottleObj(Item-Count) "GetType"
            Returning Bott-type
        If LS-type = Bott-type
            Invoke BottleObj(Item-Count) "GetCost"
                Returning Bott-cost
            IF LS-Cost = Bott-cost
                Set BottleObj(Item-Count) to BottleObj(Bottle-count)
                Set BottleObj(bottle-count) to NULL
                Subtract 1 from Bottle-Count
                Move "0" to Delete-flag
                Found-Flag
            END-IF
        END-IF
    End-Perform.
    Exit method.
END METHOD "DeleteBott".

IDENTIFICATION DIVISION.
METHOD-ID. "CalculateCost".
DATA DIVISION.
Local-Storage Section.
77  cost  Pic 999V99.
Linkage Section.
01  LS-Total-cost      Pic 9(7)V99.
PROCEDURE DIVISION Returning LS-Total-cost.
    Move 0 to LS-Total-cost
    Perform varying Item-Count from 1 by 1
        until Item-Count > Bottle-count
        Invoke bottleObj(Item-Count) "GetCost" Returning Cost
        ADD Cost      to LS-Total-cost
    End-Perform.
    Exit method.

```

```

END METHOD "CalculateCost".

IDENTIFICATION DIVISION.
METHOD-ID. "GetOrderDate".
DATA DIVISION.
Linkage Section.
01  LS-Order-Date          Pic X(8).
PROCEDURE DIVISION Returning LS-Order-date.
    Move Order-date to LS-Order-date.
    Exit method.
END METHOD "GetOrderDate".
IDENTIFICATION DIVISION.
METHOD-ID. "GetOrderNumber".
DATA DIVISION.
Linkage Section.
01  LS-Order-Number        Pic X(5).
PROCEDURE DIVISION Returning LS-Order-Number.
    Move Order-number to LS-Order-number.
    Exit method.
END METHOD "GetOrderNumber".

IDENTIFICATION DIVISION.
METHOD-ID. "scrapOrder".
DATA DIVISION.
Local-Storage Section.
PROCEDURE DIVISION.
    Subtract 1 from bottle-Count
    Perform varying Item-Count from bottle-count by -1
        until (Item-Count = 0)
            Invoke bottleObj(Item-Count) "somFree"
    End-Perform
    Exit method.
END METHOD "scrapOrder".

IDENTIFICATION DIVISION.
METHOD-ID. "InitOrder".
DATA DIVISION.
Linkage Section.
01  LS-Order-Number        Pic X(5).
01  LS-Order-Date          Pic X(8).
PROCEDURE DIVISION Using LS-Order-Number LS-Order-Date.
    Move LS-Order-Number to Order-number
    Move LS-Order-Date to Order-date
    Exit Method.
END METHOD "InitOrder".
*
*
END CLASS "WineOrder".

```

---

### D.3 User Interface Class Code

```

process test pgmname(mixed)
    IDENTIFICATION DIVISION.
    CLASS-ID. "UserInterface" Inherits SOMObject.

    ENVIRONMENT DIVISION.
    Configuration Section.
    Repository.

```

```

CLASS SOMObject IS "SOMObject"
CLASS UserInterface IS "UserInterface".

DATA DIVISION.
Working-Storage Section.
01  User-action      Pic X(10).
    88  User-add      Value "Addbott".
    88  User-delete   Value "Deletebott".
    88  User-end      Value "End".

PROCEDURE DIVISION.

IDENTIFICATION DIVISION.
METHOD-ID. "ReadAction".
DATA DIVISION.
Linkage Section.
01  Action          Pic X(10).
PROCEDURE DIVISION Returning Action.
    Display "Enter the action : add, delete, end"
    Accept action from SYSIN
    Move Function Upper-case(action) to Action
    Evaluate action
        When "ADD"
            Set User-add to TRUE
        When "DELETE"
            Set User-delete to TRUE
        When "END"
            Set User-end to TRUE
    End-evaluate
    Move User-action to action
    Exit Method.
END METHOD "ReadAction".

IDENTIFICATION DIVISION.
METHOD-ID. "ReadType".
DATA DIVISION.
Working-Storage Section.
01  WS-type         Pic X(80).
Linkage Section.
01  LS-Type         Pic X(20).
PROCEDURE DIVISION Returning LS-Type.
    Display "Enter the item"
    Accept WS-Type from SYSIN
    Move WS-Type(1:20) to LS-Type
    Exit Method.
END METHOD "ReadType".

IDENTIFICATION DIVISION.
METHOD-ID. "ReadCost".
DATA DIVISION.
Working-Storage Section.
01  WS-Cost         Pic X(6).
Linkage Section.
01  LS-Cost         Pic 999V99.
PROCEDURE DIVISION Returning LS-Cost.
    Display "Please enter the cost: "
    Accept WS-Cost from SYSIN
    Compute LS-Cost = Function Numval-c (WS-Cost)
    Exit Method.

```

```

END METHOD "ReadCost".

IDENTIFICATION DIVISION.
METHOD-ID. "WriteMessage".
DATA DIVISION.
Linkage Section.
01  LS-Flag          Pic X(4).
PROCEDURE DIVISION Using LS-Flag.
    IF LS-flag = "0"
        Display user-action " successfully completed"
    ELSE
        Display user-action " unsuccessfully completed"
    END-IF.
    Exit Method.
END METHOD "WriteMessage".

IDENTIFICATION DIVISION.
METHOD-ID. "Writeoutput".
DATA DIVISION.
Working-Storage Section.
77  Formatted-cost  Pic $Z,ZZZ,ZZ9.99.
Linkage Section.
01  Total-cost      Pic 9(7)V99.
01  Order-number    Pic 9(5).
01  Order-date      Pic X(8).
PROCEDURE DIVISION Using Total-cost Order-number Order-date.
    Move total-cost to Formatted-cost
    Display "Your order costs " Formatted-cost
    Display "Your order number is " Order-number
    Display "Your order date is " Order-date
    Exit Method.
END METHOD "Writeoutput".

IDENTIFICATION DIVISION.
METHOD-ID. "Writebottle".
DATA DIVISION.
Working-Storage Section.
01  WS-Formatted-Cost  Pic ZZ9.99.
Linkage Section.
01  LS-items.
    05 LS-count          Pic 9(4).
    05 LS-item           Occurs 1 to 64 times
                        Depending on LS-count
                        Indexed by LS-Index.
    10  LS-type          Pic X(20).
    10  LS-cost          Pic 999V99.

PROCEDURE DIVISION Using LS-items.
    Perform varying LS-Index from 1 by 1
        until  LS-Index > LS-Count
            Move  LS-Cost(LS-Index) to WS-Formatted-Cost
            Display LS-Type(LS-Index) " at " WS-Formatted-Cost
        End-Perform
    Exit Method.
END METHOD "Writebottle".

END CLASS "UserInterface".

```

---

## D.4 Bottle Class Code

```
process test pgmname(longmixed)
  IDENTIFICATION DIVISION.
  CLASS-ID. "Bottle" Inherits SOMObject.

  ENVIRONMENT DIVISION.
  Configuration Section.
  Repository.
    CLASS SOMObject IS "SOMObject"
    CLASS Bottle IS "Bottle".

  DATA DIVISION.
  Working-Storage Section.
  01 Wine-Type          Pic X(20).
  01 Wine-cost          Pic 999V99.

  IDENTIFICATION DIVISION.
  METHOD-ID. "GetType".
  ENVIRONMENT DIVISION.
  DATA DIVISION.
  Linkage Section.
  01 LS-Type            Pic X(20).
  PROCEDURE DIVISION Returning LS-Type.
    Move Wine-Type to LS-Type
    Exit method.
  END METHOD "GetType".

  IDENTIFICATION DIVISION.
  METHOD-ID. "SetType".
  ENVIRONMENT DIVISION.
  DATA DIVISION.
  Linkage Section.
  01 LS-Type            Pic X(20).
  PROCEDURE DIVISION Using LS-Type.
    Move LS-Type to Wine-Type
    Exit method.
  END METHOD "SetType".

  IDENTIFICATION DIVISION.
  METHOD-ID. "GetCost".
  ENVIRONMENT DIVISION.
  DATA DIVISION.
  Linkage Section.
  01 LS-Cost            Pic 999V99.
  PROCEDURE DIVISION Returning LS-Cost.
    Move Wine-Cost to LS-Cost
    Exit method.
  END METHOD "GetCost".

  IDENTIFICATION DIVISION.
  METHOD-ID. "SetCost".
  ENVIRONMENT DIVISION.
  DATA DIVISION.
  Linkage Section.
  01 LS-Cost            Pic 999V99.
  PROCEDURE DIVISION Using LS-Cost.
    Move LS-Cost to Wine-Cost
    Exit method.
```

```

END METHOD "SetCost".

IDENTIFICATION DIVISION.
METHOD-ID. "InitBott" .
DATA DIVISION.
Linkage Section.
01 LS-Type          Pic X(20).
01 LS-Cost          Pic 999V99.
PROCEDURE DIVISION Using LS-Type LS-Cost.
    Move LS-Type to Wine-Type
    Move LS-Cost to Wine-Cost
    Exit Method.
END METHOD "InitBott".
*
END CLASS "Bottle".

```





---

## Appendix E. Wine Store Scenario – Iteration Two Code

This appendix lists all the source modules for the second iteration of the Wine Store Scenario.

---

### E.1 Wine Client Code

```
process pgmname(mixed) test
  IDENTIFICATION DIVISION.
  PROGRAM-ID.      "Wine".

*****
*
*   The client program of the wine application does the
*   following tasks:
*   - Instantiates the UserInterface and WineOrder objects.
*   - Tells the userinterface object to read the user's
*     request.
*   - Tells the order object to process the user's request
*     and tells the userinterface object to get another
*     request until the user signals the end of the order.
*     If the request is an add or delete, sends the
*     appropriate message to the userinterface object to get
*     the item cost and type, as required by the user's
*     processing request.
*   - Tells the order object to compute the order cost.
*   - Tells the order object to get the order number.
*   - Tells the user interface object to write order cost.
*   - Tells the order to describe itself.
*   - Tells the order to write itself to the order file.
*   - Frees the objects it instantiated.
*   - Terminates.
*
*****

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
  CLASS SOMObject          IS "SOMObject"
  CLASS WineOrder          IS "WineOrder"
  CLASS Bottle             IS "WineBottle"
  CLASS FileRW             IS "FileRW"
  CLASS UserInterface      IS "UserInterface".

DATA DIVISION.
WORKING-STORAGE SECTION.

*   OBJECTS:
01  orderObj              USAGE OBJECT REFERENCE WineOrder.
01  userObj               USAGE OBJECT REFERENCE UserInterface.
01  bottleObj             USAGE OBJECT REFERENCE Bottle.
01  fileObj               USAGE OBJECT REFERENCE FileRW.

*   DATA ITEMS:
01  ACTION                PIC X(10).
01  ITEM-TYPE             PIC X(20).
```

```

01 ITEM-COST                PIC 999V99.
01 MAX-ITEMS                PIC 9(4)          COMP VALUE 64.
01 WS-PARMS.
   05 ITEM-COUNT            PIC S9(8)          COMP.
   05 WS-FLAG               PIC X.
       88 SUCCESSFUL        VALUE "0".
       88 FAILURE           VALUE "1".
01 ORDER-NUMBER             PIC 9(5).

01 ORDER-DATE               PIC X(8).
01 WS-RANDOM-VAL            PIC 9V9(5).

01 TOTAL-COST               PIC 9(7)V99.
01 WS-ITEMS.
   05 WS-COUNT              PIC S9(4).
   05 WS-ITEM               OCCURS 1 TO 64 TIMES
                           DEPENDING ON WS-COUNT
                           INDEXED BY WS-INDEX.
       10 WS-TYPE            PIC X(20).
       10 WS-COST            PIC 999V99.
EJECT

```

#### PROCEDURE DIVISION.

```

*****
*   Invoke the UserInterface class with the inherited somNew *
*   method to instantiate a userinterface object.          *
*   somNew is inherited from SOMObject.                    *
*****
      INVOKE UserInterface "somNew"    RETURNING userObj.

*****
*   We will simply use the system date for the order date   *
*   and generate a random number for the order number.      *
*   Also we will initialize the item-count field, which will *
*   control how many items are placed in the order.         *
*****
      MOVE FUNCTION CURRENT-DATE TO ORDER-DATE.

      COMPUTE WS-RANDOM-VAL = FUNCTION RANDOM.
      COMPUTE ORDER-NUMBER  = WS-RANDOM-VAL * 10000.

      MOVE      ZERO      TO  ITEM-COUNT.

*****
*   Invoke the Order class with the inherited method somNew *
*   to instantiate an order object.                          *
*****
      INVOKE WineOrder "somNew"        RETURNING  orderObj.

*****
*   Invoke the setordernumber and setorderdate methods to   *
*   set the order's date and number.                        *
*****
      INVOKE orderObj "SetOrderNumber" USING ORDER-NUMBER.
      INVOKE orderObj "SetOrderDate"   USING ORDER-DATE.

*****
*   Invoke the userinterface object with ReadAction method. *

```

```

*****
    INVOKE userObj "ReadAction" RETURNING ACTION.

*****
*   Loop until the user signals the end of the order.   *
*****
    PERFORM UNTIL ACTION = "END"
        OR ITEM-COUNT = MAX-ITEMS

        EVALUATE ACTION (1:3)
            WHEN "ADD"
*               get the type and cost from the user interface
                INVOKE userObj "ReadType" RETURNING ITEM-TYPE
                INVOKE userObj "ReadCost" RETURNING ITEM-COST

*               instantiate a bottle with those attributes
                INVOKE Bottle "somNew" RETURNING bottleObj
                INVOKE bottleObj "SetType" USING ITEM-TYPE
                INVOKE bottleObj "SetCost" USING ITEM-COST

*               add it to the collection in the order
                INVOKE orderObj "AddBottle" USING bottleObj
                    RETURNING WS-PARMS

*               if the add failed, destroy the object just
*               created because we can't do anything with it
*               and it's not in the collection.
                IF WS-FLAG = "1"
                    THEN INVOKE bottleObj "somFree"
                END-IF

*               send appropriate msg via the user interface
                INVOKE userObj "WriteMessage" USING WS-FLAG

            WHEN "DEL"
*               get the type and cost from the user interface
                INVOKE userObj "ReadType" RETURNING ITEM-TYPE
                INVOKE userObj "ReadCost" RETURNING ITEM-COST

*               create a bottle with those attributes
                INVOKE Bottle "somNew" RETURNING bottleObj
                INVOKE bottleObj "SetType" USING ITEM-TYPE
                INVOKE bottleObj "SetCost" USING ITEM-COST

*               delete copies of it from the collection
                INVOKE orderObj "RemoveBottle" USING bottleObj
                    RETURNING
                    WS-PARMS

*               destroy the bottle just created
                INVOKE bottleObj "somFree"

*               send appropriate msg via the user interface
                INVOKE userObj "WriteMessage" USING WS-FLAG

            WHEN OTHER
                CONTINUE
        END-EVALUATE
    INVOKE userObj "ReadAction" RETURNING ACTION

```

```

END-PERFORM.
*****
*   End of loop.   *
*****

*****
*****
*   Close-out processing follows.   *
*****
*****

*****
*   If no items were ordered, end the process here.   *
*****
      IF ITEM-COUNT = 0
      THEN GOBACK.

*****
*   Invoke the order object with the calculate cost method.   *
*****
      INVOKE orderObj "CalculateCost"      RETURNING TOTAL-COST.

*****
*   Invoke the order object with the getordernumber   *
*   and the getorderdate methods.   *
*****
      INVOKE orderObj "GetOrderNumber"      RETURNING ORDER-NUMBER.
      INVOKE orderObj "GetOrderDate"      RETURNING ORDER-DATE.

*****
*   Invoke the userinterface object with writeoutput method.   *
*****
      INVOKE userObj "WriteOutput"      USING TOTAL-COST
                                          ORDER-NUMBER
                                          ORDER-DATE.

*****
*   Invoke the order object with the describeorder method.   *
*****
      INVOKE orderObj "DescribeOrder" RETURNING WS-ITEMS.

*****
*   Invoke the userinterface object with writeoutput method.   *
*****
      INVOKE userObj "WriteBottle"      USING WS-ITEMS.

*****
*   Invoke the file object with the xternorder method.   *
*****
      INVOKE FileRW "somNew"      RETURNING fileObj.
      INVOKE fileObj "XternOrder" USING orderObj.

*****
*   Invoke the instantiated objects with the inherited   *
*   somDestruct method.   *
*****
      INVOKE fileObj "somFree".

      INVOKE userObj "somFree".

```

INVOKE orderObj "somFree".

```
*****
*   We're outta here...                               *
*****
GOBACK.
END PROGRAM "Wine".
```

---

## E.2 Wine Order Class Code

process pgmname(mixed) test  
IDENTIFICATION DIVISION.

```
*****
*   Class WineOrder : Inherits from SOMObject          *
*   in the SOM Class Library.                          *
*****
```

CLASS-ID. "WineOrder" INHERITS SOMObject.

```
*****
*   Class WineOrder contains the following methods:    *
*   somDefaultInit -   Initializes a WineOrder object. *
*   somFree           -   Frees bottles, collection, and order. *
*   SetOrderNumber    -   Sets the number of a WineOrder object *
*                       based on a given object reference. *
*   SetOrderDate      -   Sets the date of a WineOrder object *
*                       based on a given object reference. *
*   AddBottle         -   Adds a bottle object to the order *
*   RemoveBottle      -   Removes a bottle object from the *
*                       order. *
*   CalculateCost     -   Computes the cost of the bottle *
*                       objects in the order. *
*   DescribeOrder     -   Lists the contents of the bottles *
*                       collected in the order. *
*   GetOrderNumber    -   Retrieves the number of a WineOrder *
*                       object. *
*   GetOrderDate      -   Retrieves the date of a WineOrder *
*                       object. *
*****
```

ENVIRONMENT DIVISION.

```
*****
*   Define which classes will be used by the methods in *
*   this class.                                         *
*****
```

CONFIGURATION SECTION.

REPOSITORY.

CLASS SOMObject	IS "SOMObject"
CLASS SOMCollection	IS "somf_TSet"
CLASS SOMIterator	IS "somf_TSetIterator"
CLASS WineBottle	IS "WineBottle".

```
*****
*   Define the WineOrder Object.                       *
*****
```

```

*****

DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-EV          USAGE POINTER.

*****
*   Define the instance data of the WineOrder Object.   *
*****
01 WINEORDER-OBJECT.
   05 WINEORDER-NUMBER      PIC X(5).
   05 WINEORDER-DATE        PIC X(8).
   05 WINEORDER-LIST  USAGE OBJECT REFERENCE SOMCollection.

*****
*   Define an iterator for use on the wineorder-list.   *
*****
01 WINEORDER-ITERATOR  USAGE OBJECT REFERENCE SOMIterator.
   EJECT

PROCEDURE DIVISION.

*****
*****
*   The overridden method somDefaultInit initializes the *
*   WineOrder instance, and creates the collection to be *
*   used in the order.                                   *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "somDefaultInit"      OVERRIDE.

DATA DIVISION.

PROCEDURE DIVISION.

*****
*   Initialize the SOM global environment variable.   *
*****
CALL "somGetGlobalEnvironment" RETURNING WS-EV.
*****
*   Now initialize an empty collection for us to add bottles *
*   into with the addBottle method.                     *
*****
INVOKE SOMCollection "somNew"
      RETURNING WINEORDER-LIST.

*****
*   Instantiate an iterator object and associate it with the *
*   collection.                                             *
*****
INVOKE WINEORDER-LIST "somfCreateIterator"
      USING      BY VALUE WS-EV
      RETURNING WINEORDER-ITERATOR.

*****
*   EXIT and END the method.                               *
*****
EXIT METHOD.
END METHOD "somDefaultInit".

```

```

EJECT
*****
*****
*   The overridden method somFree      destroys the bottle      *
*   objects created, the collection object, and the order      *
*   object.                                                     *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "somFree"          OVERRIDE.

DATA DIVISION.
WORKING-STORAGE SECTION.
01 CollectedBottle          USAGE OBJECT REFERENCE WineBottle.
01 ITEM-COUNT                PIC S9(8)    COMP.

PROCEDURE DIVISION.

*****
*   Get the collected objects.                                     *
*****
        INVOKE WINEORDER-LIST "somfDeleteAll"
                USING          BY VALUE WS-EV.

*****
*   Free the list and iterator objects                             *
*****
        INVOKE WINEORDER-ITERATOR "somFree".

        INVOKE WINEORDER-LIST "somFree".

*****
*   Free thyself...Use SUPER so we don't recurse back into      *
*   this method.                                                 *
*****
        INVOKE SUPER "somFree".

*****
*   EXIT and END the method.                                     *
*****
        EXIT METHOD.
        END METHOD "somFree".
EJECT
*****
*****
*   Method GetOrderNumber gets the number of WineOrder based   *
*   on the object reference of the WineOrder.                  *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "GetOrderNumber".

DATA DIVISION.
WORKING-STORAGE SECTION.

*****
*   Define the linkage attributes.                                 *
*****
LINKAGE SECTION.
01 LS-ORDERNUMBER          PIC X(5).

```





```

*****
MOVE LS-ORDERNUMBER TO WINEORDER-NUMBER.

EXIT METHOD.
END METHOD "SetOrderNumber".
EJECT
*****
*****
*   Method SetOrderDate sets the date of a WineOrder based *
*   on the object reference of the WineOrder.               *
*****

IDENTIFICATION DIVISION.
METHOD-ID. "SetOrderDate".

DATA DIVISION.
WORKING-STORAGE SECTION.

*****
*   Define the linkage attributes.                             *
*****

LINKAGE SECTION.
01 LS-ORDERDATE                                PIC X(8).

PROCEDURE DIVISION                                USING      LS-ORDERDATE.

*****
*   Move data from the LINKAGE SECTION.                       *
*****

MOVE LS-ORDERDATE TO WINEORDER-DATE.

EXIT METHOD.
END METHOD "SetOrderDate".
EJECT
*****
*****
*   Method DescribeOrder describes the order contents.       *
*****

IDENTIFICATION DIVISION.
METHOD-ID. "DescribeOrder".

DATA DIVISION.

LOCAL-STORAGE SECTION.
01 CollectedBottle    USAGE OBJECT REFERENCE WineBottle.
01 WS-TYPE              PIC X(20).
01 WS-COST              PIC 999V99.
01 ITEM-COUNT           PIC S9(8)  COMP.

LINKAGE SECTION.
01 LS-ITEMS.
   05 LS-ITEM-COUNT      PIC S9(4).
   05 LS-ITEM            OCCURS 1 TO 64 TIMES
                        DEPENDING ON LS-ITEM-COUNT
                        INDEXED BY  LS-INDEX.
   10 LS-TYPE            PIC X(20).
   10 LS-COST            PIC 999V99.

PROCEDURE DIVISION                                RETURNING LS-ITEMS.

```

```

*****
*   Get the count of the number of items in the collection.   *
*****
        INVOKE WINEORDER-LIST "somfCount"
                USING      BY VALUE WS-EV
                RETURNING   ITEM-COUNT.
        MOVE ITEM-COUNT TO LS-ITEM-COUNT.

*****
*   Get the first one in the collection.                       *
*****
        IF ITEM-COUNT > 0
            THEN SET LS-INDEX TO 1
                INVOKE WINEORDER-ITERATOR "somfFirst"
                        USING      BY VALUE WS-EV
                        RETURNING CollectedBottle
                PERFORM GET-TYPE-N-COST
        END-IF.

*****
*   Get the rest...                                           *
*****
        SUBTRACT 1 FROM ITEM-COUNT.
        IF ITEM-COUNT > 0
            THEN PERFORM ITEM-COUNT TIMES
                SET LS-INDEX UP BY 1
                INVOKE WINEORDER-ITERATOR "somfNext"
                        USING      BY VALUE WS-EV
                        RETURNING CollectedBottle
                PERFORM GET-TYPE-N-COST
            END-PERFORM
        END-IF.

*****
*   Exit and end the method.                                   *
*****
        EXIT METHOD.

*****
*   Invoke the gettype and getcost methods on the bottle     *
*   object and move the returned attributes to the table.     *
*****
        GET-TYPE-N-COST.
            INVOKE CollectedBottle "GetType" RETURNING WS-TYPE.
            MOVE WS-TYPE TO LS-TYPE (LS-INDEX).
            INVOKE CollectedBottle "GetCost" RETURNING WS-COST.
            MOVE WS-COST TO LS-COST (LS-INDEX).

        END METHOD "DescribeOrder".
        EJECT

*****
*   Method CalculateCost computes the cost of the order.      *
*****
        IDENTIFICATION DIVISION.
        METHOD-ID. "CalculateCost".

        DATA DIVISION.
        WORKING-STORAGE SECTION.

```

```

01 CollectedBottle    USAGE OBJECT REFERENCE WineBottle.
01 ITEM-COUNT          PIC S9(8)  COMP.
01 WS-COST             PIC 999V99.

*****
*   Define the linkage attributes.                               *
*****

LINKAGE SECTION.
01 LS-COST             PIC 9(7)V99.

PROCEDURE DIVISION
    RETURNING  LS-COST.

*****
*   Initialize the accumulator for the total cost.              *
*****

    MOVE ZERO TO LS-COST.

*****
*   Get the count of the number of items in the collection.    *
*****

    INVOKE WINEORDER-LIST "somfCount"
        USING      BY VALUE WS-EV
        RETURNING ITEM-COUNT.

*****
*   Get the first one in the collection.                        *
*****

    IF ITEM-COUNT > 0
        INVOKE WINEORDER-ITERATOR "somfFirst"
            USING      BY VALUE WS-EV
            RETURNING CollectedBottle
        PERFORM GET-COST
    END-IF.

*****
*   Get the rest...                                            *
*****

    SUBTRACT 1 FROM ITEM-COUNT.
    IF ITEM-COUNT > 0
        THEN PERFORM ITEM-COUNT TIMES
            INVOKE WINEORDER-ITERATOR "somfNext"
                USING      BY VALUE WS-EV
                RETURNING CollectedBottle
            PERFORM GET-COST
        END-PERFORM
    END-IF.

*****
*   EXIT the method and return.                                *
*****

    EXIT METHOD.

*****
*   Invoke the getcost method on the bottle object and         *
*   accumulate the cost.                                       *
*****

    GET-COST.
    INVOKE CollectedBottle "GetCost" RETURNING WS-COST.
    ADD WS-COST TO LS-COST.

```

```

END METHOD "CalculateCost".
EJECT
*****
*****
*   Method AddBottle adds a bottle of wine to the bottle   *
*   collection in the wine order.                           *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "AddBottle".

DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-BEFORE-COUNT          PIC S9(8)    COMP.
01 WS-AFTER-COUNT          PIC S9(8)    COMP.
01 CollectedBottle        USAGE OBJECT REFERENCE WineBottle.

01 theEqualFlag            PIC X.
01 ITEM-FOUND-FLAG         PIC X.
01 ITEM-COUNT              PIC S9(8)    COMP.
01 LOOP-COUNT              PIC S9(8)    COMP.

*****
*   Define the linkage attributes.                           *
*****
LINKAGE SECTION.
01 LS-BOTTLE                USAGE OBJECT REFERENCE WineBottle.
01 LS-PARMS.
   05 LS-ITEM-COUNT         PIC S9(8)    COMP.
   05 LS-FLAG               PIC X.

PROCEDURE DIVISION
                                USING      LS-BOTTLE
                                RETURNING   LS-PARMS.

                                MOVE LOW-VALUE      TO ITEM-FOUND-FLAG.

*****
*   Get the count of items before adding the object.        *
*****
                                INVOKE WINEORDER-LIST "somfCount"
                                                USING      BY VALUE WS-EV
                                                RETURNING WS-BEFORE-COUNT.
                                MOVE      WS-BEFORE-COUNT TO ITEM-COUNT.

*****
*   Get the first one in the collection.                     *
*****
                                IF ITEM-COUNT NOT = 0
                                    THEN INVOKE WINEORDER-ITERATOR "somfFirst"
                                                USING      BY VALUE WS-EV
                                                RETURNING CollectedBottle
                                    PERFORM CHECK-EQUAL
                                END-IF.

*****
*   Get the rest...                                         *
*****
                                SUBTRACT 1 FROM ITEM-COUNT.
                                IF ITEM-COUNT > 0

```

```

        THEN PERFORM VARYING LOOP-COUNT
            FROM 1 BY 1
            UNTIL LOOP-COUNT      > ITEM-COUNT
                OR ITEM-FOUND-FLAG = HIGH-VALUE
            INVOKE WINEORDER-ITERATOR "somfNext"
                USING      BY VALUE WS-EV
                RETURNING CollectedBottle
            PERFORM CHECK-EQUAL
        END-PERFORM
    END-IF.

*****
*   Add the bottle to the collection if it hasn't been   *
*   added before.                                       *
*****
        IF ITEM-FOUND-FLAG = LOW-VALUE
            THEN INVOKE WINEORDER-LIST "somfAdd"
                USING BY VALUE WS-EV
                BY VALUE LS-BOTTLE.

*****
*   Get the count of items after adding the object.     *
*****
        INVOKE WINEORDER-LIST "somfCount"
            USING      BY VALUE WS-EV
            RETURNING WS-AFTER-COUNT.
        MOVE WS-AFTER-COUNT    TO LS-ITEM-COUNT.

*****
*   If the counts are the same the add failed.           *
*****
        IF WS-BEFORE-COUNT = WS-AFTER-COUNT
            THEN MOVE "1" TO LS-FLAG
        ELSE
            MOVE "0" TO LS-FLAG
        END-IF.

*****
*   EXIT the method and return.                         *
*****
        EXIT METHOD.

*****
*   Invoke the somfIsEqual method in the bottle object to *
*   see if the objects are equal. Set a flag if they are. *
*****
        CHECK-EQUAL.
        INVOKE CollectedBottle "somfIsEqual"
            USING      BY VALUE WS-EV
            BY VALUE LS-BOTTLE
            RETURNING theEqualFlag.
        IF theEqualFlag = HIGH-VALUE
            THEN MOVE HIGH-VALUE TO ITEM-FOUND-FLAG.

    END METHOD "AddBottle".
    EJECT
*****
*   Method RemoveBottle removes a bottle from the bottle *
*****

```

```

*      collection in the wine order.                                     *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "RemoveBottle".

DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-BEFORE-COUNT          PIC S9(8)      COMP.
01 WS-AFTER-COUNT           PIC S9(8)      COMP.
01 CollectedBottle         USAGE OBJECT REFERENCE WineBottle.
01 theEqualFlag             PIC X.
01 ITEM-COUNT               PIC S9(8)      COMP.
01 LOOP-COUNT               PIC S9(8)      COMP.

*****
*      Define the linkage attributes.                                   *
*****
LINKAGE SECTION.
01 LS-BOTTLE                USAGE OBJECT REFERENCE WineBottle.
01 LS-PARMS.
   05 LS-ITEM-COUNT         PIC S9(8)      COMP.
   05 LS-FLAG               PIC X.

PROCEDURE DIVISION
                                USING      LS-BOTTLE
                                RETURNING   LS-PARMS.

*****
*      Get the count of items before the delete.                       *
*****
      INVOKE WINEORDER-LIST "somfCount"
                                USING      BY VALUE WS-EV
                                RETURNING   WS-BEFORE-COUNT.
      MOVE WS-BEFORE-COUNT TO ITEM-COUNT.

*****
*      Get the first one in the collection.                             *
*****
      IF ITEM-COUNT NOT = 0
          THEN INVOKE WINEORDER-ITERATOR "somfFirst"
                                USING      BY VALUE WS-EV
                                RETURNING   CollectedBottle
          PERFORM CHECK-EQUAL-N-REMOVE
      END-IF.

*****
*      Get the rest...                                                 *
*****
      SUBTRACT 1 FROM ITEM-COUNT.
      IF ITEM-COUNT > 0
          THEN PERFORM VARYING LOOP-COUNT
                                FROM 1 BY 1
                                UNTIL LOOP-COUNT > ITEM-COUNT
                                OR theEqualFlag = HIGH-VALUE
          INVOKE WINEORDER-ITERATOR "somfNext"
                                USING      BY VALUE WS-EV
                                RETURNING   CollectedBottle
          PERFORM CHECK-EQUAL-N-REMOVE
      END-PERFORM

```

```

END-IF.

*****
*   Get the count of items after the delete.   *
*****
    INVOKE WINEORDER-LIST "somfCount"
                USING      BY VALUE WS-EV
                RETURNING WS-AFTER-COUNT.
    MOVE WS-AFTER-COUNT   TO LS-ITEM-COUNT.

*****
*   If the counts are the same the delete failed.   *
*****
    IF WS-BEFORE-COUNT = WS-AFTER-COUNT
        THEN MOVE "1" TO LS-FLAG
    ELSE
        MOVE "0" TO LS-FLAG
    END-IF.

*****
*   EXIT the method and return.   *
*****
    EXIT METHOD.

CHECK-EQUAL-N-REMOVE.
    INVOKE CollectedBottle "somfIsEqual"
                USING      BY VALUE WS-EV
                BY VALUE LS-BOTTLE
                RETURNING theEqualFlag.
*****
*   If we find one, remove it from the list.   *
*****
    IF theEqualFlag = HIGH-VALUE
        THEN INVOKE WINEORDER-LIST "somfRemove"
                USING BY VALUE WS-EV
                BY VALUE CollectedBottle
        INVOKE CollectedBottle "somFree".

    END METHOD "RemoveBottle".
    SKIP3
    SKIP3

*****
*   End object definition and class WineOrder.   *
*****
    END CLASS "WineOrder".

```

---

### E.3 User Interface Class Code

```

process pgmname(mixed) test
    IDENTIFICATION DIVISION.

*****
*   Class UserInterface: Inherits from SOMObject   *
*   in the SOM Class Library.   *
*****

    CLASS-ID. "UserInterface" INHERITS SOMObject.

```

```

*****
*   Class UserInterface contains the following methods:   *
*   ReadAction      -   Gets the input command from the   *
*                       system user.                       *
*   ReadType        -   Gets the type of item from the     *
*                       system user.                       *
*   ReadCost        -   Gets the cost of item from the     *
*                       system user.                       *
*   WriteMessage    -   Displays a system status message to *
*                       the system user.                   *
*   WriteOutput     -   Displays the cost of the order and  *
*                       order to the system user.          *
*   WriteBottle     -   Displays the attributes of a bottle *
*                       collected in the order.            *
*****

```

#### ENVIRONMENT DIVISION.

```

*****
*   Define which classes will be used by the methods in   *
*   this class.                                           *
*****

```

#### CONFIGURATION SECTION.

##### REPOSITORY.

CLASS SOMObject IS "SOMObject".

```

*****
*   Define the UserInterface Object.                     *
*****

```

#### DATA DIVISION.

##### WORKING-STORAGE SECTION.

```

*****
*   Define the instance data of the UserInterface Object. *
*****
01 USER-ACTION          PIC X(10).
   88 UA-ADD             VALUE "Add".
   88 UA-DELETE          VALUE "Delete".
   88 UA-END             VALUE "End".
   EJECT

```

#### PROCEDURE DIVISION.

```

*****
*****
*   Method ReadAction gets the system user's command to be *
*   processed.                                             *
*****

```

#### IDENTIFICATION DIVISION.

##### METHOD-ID. "ReadAction".

#### DATA DIVISION.

##### WORKING-STORAGE SECTION.

01 WS-EDIT-FLAG PIC X.

```

*****

```



```

*      Define the linkage attributes.                                     *
*****
LINKAGE SECTION.
01  LS-ACTION                                PIC X(10).

PROCEDURE DIVISION                                RETURNING    LS-ACTION.

    MOVE LOW-VALUE TO WS-EDIT-FLAG.
    PERFORM UNTIL WS-EDIT-FLAG NOT = LOW-VALUE
        DISPLAY "Enter the action desired:  add, delete, end: "
        ACCEPT USER-ACTION                                FROM SYSIN
        MOVE FUNCTION UPPER-CASE (USER-ACTION) TO USER-ACTION
        MOVE USER-ACTION                                TO LS-ACTION

        EVALUATE USER-ACTION (1:3)
            WHEN "ADD"
                MOVE HIGH-VALUE TO WS-EDIT-FLAG
            WHEN "DEL"
                MOVE HIGH-VALUE TO WS-EDIT-FLAG
            WHEN "END"
                MOVE HIGH-VALUE TO WS-EDIT-FLAG
            WHEN OTHER
                DISPLAY "Requested action was " USER-ACTION
                DISPLAY "Try again, fumblefingers!!!"
        END-EVALUATE
    END-PERFORM.
    EXIT METHOD.
END METHOD "ReadAction".
EJECT
*****
*      Method ReadType gets the type of item to be processed.         *
*****

IDENTIFICATION DIVISION.
METHOD-ID. "ReadType".

DATA DIVISION.
WORKING-STORAGE SECTION.
01  WS-TYPE                                PIC X(80).

*****
*      Define the linkage attributes.                                     *
*****
LINKAGE SECTION.
01  LS-TYPE                                PIC X(20).

PROCEDURE DIVISION                                RETURNING    LS-TYPE.

    DISPLAY "Enter the item: ".
    ACCEPT WS-TYPE                                FROM SYSIN.
    MOVE FUNCTION UPPER-CASE (WS-TYPE) TO    WS-TYPE.
    MOVE    WS-TYPE (1:20)                TO    LS-TYPE.
    EXIT METHOD.
END METHOD "ReadType".
EJECT
*****
*      Method ReadCost gets the cost of the item to be processed.*
*****

```

```
IDENTIFICATION DIVISION.
METHOD-ID. "ReadCost".
```

```
DATA DIVISION.
WORKING-STORAGE SECTION.
```

```
01 WS-EDIT-FLAG          PIC X.
01 WS-COST-WORK          PIC X(6).
```

```
*****
*   Define the linkage attributes.   *
*****
```

```
LINKAGE SECTION.
```

```
01 LS-COST              PIC 999V99.
```

```
PROCEDURE DIVISION          RETURNING    LS-COST.
```

```
    MOVE LOW-VALUE TO WS-EDIT-FLAG.
    PERFORM UNTIL WS-EDIT-FLAG = HIGH-VALUE
        DISPLAY "Enter the cost: "
        ACCEPT WS-COST-WORK          FROM SYSIN
        COMPUTE LS-COST = FUNCTION NUMVAL (WS-COST-WORK)
        IF LS-COST NUMERIC
            THEN MOVE HIGH-VALUE TO WS-EDIT-FLAG
        ELSE
            DISPLAY "Cost is not numeric - try again "
            DISPLAY "and get it right this time!!!"
        END-IF
    END-PERFORM.
    EXIT METHOD.
END METHOD "ReadCost".
EJECT
```

```
*****
*****
*   Method WriteMessage lets the system user know if the   *
*   requested action was successful.                         *
*****
```

```
IDENTIFICATION DIVISION.
METHOD-ID. "WriteMessage".
```

```
DATA DIVISION.
```

```
*****
*   Define the linkage attributes.   *
*****
```

```
LINKAGE SECTION.
```

```
01 LS-FLAG              PIC X.
```

```
PROCEDURE DIVISION          USING        LS-FLAG.
```

```
    IF LS-FLAG = "0"
        THEN DISPLAY USER-ACTION "successful "
    ELSE
        DISPLAY USER-ACTION "failed "
    END-IF.
    EXIT METHOD.
END METHOD "WriteMessage".
EJECT
```

```
*****
*****
```

```

*      Method WriteOutput displays the order number and cost      *
*      to the system user.                                         *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "WriteOutput".

DATA DIVISION.
WORKING-STORAGE SECTION.
01 FORMATTED-COST          PIC $Z,ZZZ,ZZ9.99.

*****
*      Define the linkage attributes.                               *
*****
LINKAGE SECTION.
01 LS-TOTAL-COST          PIC 9(7)V99.
01 LS-ORDER-NUMBER        PIC 9(5).
01 LS-ORDER-DATE          PIC X(8).

PROCEDURE DIVISION
      USING      LS-TOTAL-COST
                LS-ORDER-NUMBER
                LS-ORDER-DATE.

      MOVE LS-TOTAL-COST TO FORMATTED-COST.
      DISPLAY "Your order number " LS-ORDER-NUMBER
              " placed on "        LS-ORDER-DATE
              " costs "           FORMATTED-COST.
      EXIT METHOD.
END METHOD "WriteOutput".
EJECT
*****
*****
*      Method WriteBottle displays the attributes of bottles      *
*      that have been collected in the order.                     *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "WriteBottle".

DATA DIVISION.

WORKING-STORAGE SECTION.

01 WS-FORMATTED-COUNT      PIC ZZ9.
01 WS-FORMATTED-COST      PIC ZZ9.99.

*****
*      Define the linkage attributes.                               *
*****
LINKAGE SECTION.
01 LS-ITEMS.
   05 LS-ITEM-COUNT        PIC S9(4).
   05 LS-ITEM              OCCURS 1 TO 64 TIMES
                           DEPENDING ON LS-ITEM-COUNT
                           INDEXED BY   LS-INDEX.
   10 LS-TYPE              PIC X(20).
   10 LS-COST              PIC 999V99.

PROCEDURE DIVISION
      USING      LS-ITEMS.

      MOVE LS-ITEM-COUNT TO WS-FORMATTED-COUNT.

```

```

        DISPLAY "Contains " WS-FORMATTED-COUNT " items".
        PERFORM VARYING LS-INDEX FROM 1 BY 1
            UNTIL LS-INDEX > LS-ITEM-COUNT
                MOVE LS-COST (LS-INDEX) TO WS-FORMATTED-COST
                DISPLAY LS-TYPE (LS-INDEX)" @ " WS-FORMATTED-COST
        END-PERFORM.
        EXIT METHOD.
    END METHOD "WriteBottle".
    SKIP3
    SKIP3
*****
*   End object definition and class UserInterface.           *
*****
END CLASS "UserInterface".

```

---

## E.4 Bottle Class Code

```

process pgmname(mixed) test
    IDENTIFICATION DIVISION.

*****
*   Class WineBottle : Inherits from somf_MCollectible      *
*   in the SOM Class Library.                               *
*****

    CLASS-ID. "WineBottle" INHERITS somf-MCollectible.

*****
*   Class WineBottle contains the following methods:       *
*   somfIsEqual - Provides SOM a method to see if two      *
*   objects are equivalent.                                 *
*   SetCost - Sets the cost of a WineBottle object         *
*   based on a given object reference.                      *
*   SetType - Sets the type of a WineBottle object         *
*   based on a given object reference.                      *
*   GetCost - Retrieves the cost of a WineBottle           *
*   object based on a given object reference.              *
*   GetType - Retrieves the type of a WineBottle           *
*   object based on a given object reference.              *
*****

    ENVIRONMENT DIVISION.

*****
*   Define which classes will be used by the methods in    *
*   this class.                                             *
*****

    CONFIGURATION SECTION.
    REPOSITORY.
        CLASS WineBottle IS "WineBottle"
        CLASS somf-MCollectible IS "somf_MCollectible".

*****
*   Define the WineBottle Object.                           *
*****

```

```

*****
*   Define the instance data of the WineBottle Object.   *
*****
01  WINEBOTTLE-OBJECT.
    05  WINE-TYPE                PIC X(20).
    05  WINE-COST                PIC 999V99.
    EJECT

PROCEDURE DIVISION.

*****
*****
*   Method somfIsEqual provides SOM a method to see if two *
*   bottle objects are equivalent.  In our case, if their  *
*   types and costs are the same, we consider them equal.  *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "somfIsEqual"        OVERRIDE.

DATA DIVISION.
LOCAL-STORAGE SECTION.
01  ITEMTYPE                PIC X(20).
01  ITEMCOST                PIC 999V99.

LINKAGE SECTION.
01  LS-EV                  USAGE POINTER.
01  theBottle              Usage Object Reference WineBottle.
01  theEqualFlag           PIC X.

PROCEDURE DIVISION                USING BY VALUE LS-EV
                                   BY VALUE theBottle
                                   RETURNING      theEqualFlag.

*****
*   Get the type and cost of the bottle object   *
*****
    INVOKE theBottle    "GetType"    RETURNING ITEMTYPE.
    INVOKE theBottle    "GetCost"    RETURNING ITEMCOST.

*****
*   Get those just obtained to the attributes of this *
*   instance.  If they are equal, set the equality flag *
*   to a binary 1, else set it to a low-value.        *
*****
    IF (WINE-TYPE = ITEMTYPE) AND
       (WINE-COST = ITEMCOST)
       THEN MOVE HIGH-VALUE TO theEqualFlag
    ELSE
       MOVE LOW-VALUE TO theEqualFlag.

    EXIT METHOD.
    END METHOD "somfIsEqual".
    EJECT
*****
*****

```

```

*      Method GetType Gets the type of a WineBottle based on the *
*      object reference of the WineBottle.                        *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "GetType".

DATA DIVISION.
WORKING-STORAGE SECTION.

*****
*      Define the linkage attributes.                             *
*****
LINKAGE SECTION.
01  LS-TYPE                                PIC X(20).

PROCEDURE DIVISION                                RETURNING  LS-TYPE.

*****
*      Move data to the LINKAGE SECTION.                         *
*****
MOVE WINE-TYPE TO LS-TYPE.

EXIT METHOD.
END METHOD "GetType".
EJECT
*****
*      Method GetCost Gets the COST of a WineBottle based on the *
*      object reference of the WineBottle.                       *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "GetCost".

DATA DIVISION.
WORKING-STORAGE SECTION.

*****
*      Define the linkage attributes.                             *
*****
LINKAGE SECTION.
01  LS-COST                                PIC 999V99.

PROCEDURE DIVISION                                RETURNING  LS-COST.

*****
*      Move data to the LINKAGE SECTION.                         *
*****
MOVE WINE-COST TO LS-COST.

EXIT METHOD.
END METHOD "GetCost".
EJECT
*****
*      Method SetType Sets the type of a WineBottle based on the *
*      object reference of the WineBottle.                       *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "SetType".

```

```
DATA DIVISION.
WORKING-STORAGE SECTION.
```

```
*****
*   Define the linkage attributes.                               *
*****
```

```
LINKAGE SECTION.
```

```
01 LS-TYPE                                PIC X(20).
```

```
PROCEDURE DIVISION                        USING      LS-TYPE.
```

```
*****
*   Move data to the LINKAGE SECTION.                           *
*****
```

```
MOVE LS-TYPE TO WINE-TYPE.
```

```
EXIT METHOD.
```

```
END METHOD "SetType".
```

```
EJECT
```

```
*****
*****
*   Method SetCost Sets the COST of a WineBottle based on the *
*   object reference of the WineBottle.                       *
*****
```

```
IDENTIFICATION DIVISION.
```

```
METHOD-ID. "SetCost".
```

```
DATA DIVISION.
WORKING-STORAGE SECTION.
```

```
*****
*   Define the linkage attributes.                               *
*****
```

```
LINKAGE SECTION.
```

```
01 LS-COST                                PIC 999V99.
```

```
PROCEDURE DIVISION                        USING      LS-COST.
```

```
*****
*   Move data to the LINKAGE SECTION.                           *
*****
```

```
MOVE LS-COST TO WINE-COST.
```

```
EXIT METHOD.
```

```
END METHOD "SetCost".
```

```
SKIP3
```

```
SKIP3
```

```
*****
*   End object definition and class WineBottle.               *
*****
```

```
END CLASS "WineBottle".
```

---

## E.5 FileRW Class Code

```
process pgmname(mixed) test
  IDENTIFICATION DIVISION.

  *****
  *   Class FileRW   : Inherits from SOMObject           *
  *                   in the SOM Class Library.           *
  *****

  CLASS-ID.  "FileRW"  INHERITS SOMObject.

  *****
  *   Class FileRW contains the following methods:       *
  *   XternOrder    -   Externalizes an order to a flat   *
  *                   file.                               *
  *****

  ENVIRONMENT DIVISION.

  *****
  *   Define which classes will be used by the methods in *
  *   this class.                                         *
  *****

  CONFIGURATION SECTION.
  REPOSITORY.
      CLASS SOMObject          IS "SOMObject"
      CLASS WineOrder          IS "WineOrder".

  *****
  *   Define the FileRW object.                           *
  *****

  DATA DIVISION.
  WORKING-STORAGE SECTION.

  PROCEDURE DIVISION.

  *****
  *   Method XternOrder writes the order to a flat file.  *
  *****

  IDENTIFICATION DIVISION.
  METHOD-ID. "XternOrder".

  ENVIRONMENT DIVISION.
  INPUT-OUTPUT SECTION.
  FILE-CONTROL.
      SELECT ORDERS          ASSIGN TO   ORDERS
      FILE STATUS IS WS-STATUS-FLAG
      ORGANIZATION IS LINE SEQUENTIAL.

  DATA DIVISION.
  FILE SECTION.
  FD ORDERS EXTERNAL
      RECORD CONTAINS 255.
  01 ORDER-RECORD          PIC X(255).
```



```

WORKING-STORAGE SECTION.
01 WS-STATUS-FLAG          PIC XX.
01 WS-EV                   USAGE POINTER.
01 WS-ORDER-RECORD.
    05 WS-ORDER-NUMBER      PIC X(5).
    05 WS-ORDER-DATE        PIC X(8).
    05 FILLER               PIC XXX.
    05 WS-ITEMS.
        10 WS-ORDER-COUNT    PIC S9(4).
        10 WS-ORDER-ITEM     OCCURS 1 TO 64
                               DEPENDING ON WS-ORDER-COUNT
                               INDEXED BY WS-INDEX.
        15 WSO-TYPE          PIC X(20).
        15 WSO-COST          PIC 999V99.

LINKAGE SECTION.
01 orderObj                USAGE OBJECT REFERENCE WineOrder.

PROCEDURE DIVISION        USING orderObj.

*****
*   Open the flat file for output.   *
*****
    OPEN OUTPUT ORDERS.
    MOVE SPACES TO WS-ORDER-RECORD.

    INVOKE orderObj "GetOrderNumber" RETURNING WS-ORDER-NUMBER.
    INVOKE orderObj "GetOrderDate"   RETURNING WS-ORDER-DATE.
    INVOKE orderObj "DescribeOrder"  RETURNING WS-ITEMS.

*****
*   Write the record.               *
*****
    WRITE ORDER-RECORD FROM WS-ORDER-RECORD.

*****
*   Close the order file after writing the record to it.   *
*****
    CLOSE ORDERS.

    EXIT METHOD.
END METHOD "XternOrder".
SKIP3
SKIP3
*****
*   End object definition and class FileRW.   *
*****
END CLASS "FileRW".

```



---

## Appendix F. Wine Store Scenario – Iteration Three Code

This appendix lists all the source modules for the third iteration of the Wine Store Scenario.

---

### F.1 Wine Client Code

```
process pgmname(longmixed) test
  IDENTIFICATION DIVISION.
  PROGRAM-ID. "Wine".

  *****
  *
  *   The client program of the wine application does the
  *   following tasks:
  *   - Instantiates the UserInterface obejct.
  *   - If a status request:
  *     -- instantiates an OldOrder object
  *     -- invokes the UserInterface object to get the order
  *       number
  *     -- invokes the OldOrder to report out-of-stock items
  *     -- invokes the UserInterface object to display the
  *       status of the out-of-stock items
  *     -- Frees the OldOrder object
  *   - If a request for a new order:
  *     -- Tells the order object to process the user's request
  *       and tells the userinterface object to get another
  *       request until the user signals the end of the order.
  *       If the request is an add or delete, sends the
  *       appropriate message to the userinterface object for
  *       the item cost and type, as required by the user's
  *       processing request.
  *     -- Tells the order object to compute the order cost.
  *     -- Tells the order object to get the order number.
  *     -- Tells the user interface object to write order cost.
  *     -- Tells the order to describe itself.
  *     -- Tells the order to write itself to the order file.
  *     -- Frees the objects it instantiated.
  *   - Frees the UserInterface object.
  *   - Terminates.
  *
  *****

  ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  REPOSITORY.
    CLASS SOMObject          IS "SOMObject"
    CLASS WineOrder          IS "WineOrder"
    CLASS OldOrder           IS "OldOrder"
    CLASS Bottle             IS "WineBottle"
    CLASS FileRW             IS "FileRW"
    CLASS UserInterface      IS "UserInterface".

  DATA DIVISION.
  WORKING-STORAGE SECTION.
```

```

*   OBJECTS:
01  orderObj           USAGE OBJECT REFERENCE WineOrder.
01  oldOrderObj        USAGE OBJECT REFERENCE OldOrder.
01  userObj            USAGE OBJECT REFERENCE UserInterface.
01  bottleObj          USAGE OBJECT REFERENCE Bottle.
01  fileObj            USAGE OBJECT REFERENCE FileRW.

*   DATA ITEMS:
01  ACTION              PIC X(10).
01  PROCESS              PIC X(10).
01  ITEM-TYPE            PIC X(20).
01  ITEM-COST            PIC 999V99.
01  MAX-ITEMS            PIC 9(4)          COMP VALUE 64.
01  WS-PARMS.
    05  ITEM-COUNT        PIC S9(8)          COMP.
    05  WS-FLAG           PIC X.
        88  SUCCESSFUL      VALUE "0".
        88  FAILURE         VALUE "1".

01  ORDER-DATE           PIC X(8).

01  WS-RANDOM-VAL        PIC 9V9(5).
01  ORDER-NUMBER         PIC 9(5).
01  TOTAL-COST           PIC 9(7)V99.
01  WS-ITEMS.
    05  WS-COUNT          PIC S9(4).
    05  WS-ITEM           OCCURS 1 TO 64 TIMES
                          DEPENDING ON WS-COUNT
                          INDEXED BY WS-INDEX.
        10  WS-TYPE        PIC X(20).
        10  WS-COST        PIC 999V99.

01  OUT-ITEMS.
    05  OUT-COUNT         PIC S9(4).
    05  OUT-ITEM          OCCURS 1 TO 64 TIMES
                          DEPENDING ON OUT-COUNT
                          INDEXED BY OUT-INDEX.
        10  OUT-TYPE       PIC X(20).
        10  OUT-COST       PIC 999V99.

01  WS-ORDER-RECORD.
    05  WSO-ORDER-NUMBER  PIC X(5).
    05  WSO-ORDER-DATE    PIC X(8).
    05  FILLER            PIC XXX.
    05  WSO-ITEMS.
        10  WSO-ORDER-COUNT PIC S9(4).
        10  WSO-ORDER-ITEM OCCURS 1 TO 64
                          DEPENDING ON WSO-ORDER-COUNT
                          INDEXED BY WSO-INDEX.
            15  WSO-TYPE    PIC X(20).
            15  WSO-COST    PIC 999V99.

EJECT

```

PROCEDURE DIVISION.

```

*****
*   Invoke the UserInterface class with the inherited somNew *
*   method to instantiate a userinterface object.           *
*   somNew is inherited from SOMObject.                      *
*****

```

```

*****
        INVOKE UserInterface "somNew"    RETURNING userObj.

*****
*   Invoke the UserInterface class with the ReadProcess      *
*   method to obtain the process desired by the system user. *
*****
        INVOKE userObj    "ReadProcess"    RETURNING PROCESS.

*****
*   Use the process to control the path through this program. *
*****
        PERFORM UNTIL PROCESS (1:4) = "EXIT"
            EVALUATE PROCESS (1:3)
                WHEN "STA"
                    PERFORM CHECK-OLD-ORDER THRU CHECK-EXIT
                WHEN "NEW"
                    PERFORM CREATE-NEW-ORDER THRU CREATE-EXIT
                WHEN OTHER
                    CONTINUE
            END-EVALUATE
            INVOKE userObj "ReadProcess"    RETURNING PROCESS
        END-PERFORM.

*****
*   Free the user interface object we created.                *
*****
        INVOKE userObj    "somFree".

*****
*   We're outta here...                                       *
*****
        GOBACK.
        EJECT
        CHECK-OLD-ORDER.
*****
*   Invoke the old order class with the inherited method      *
*   somNew to instantiate an old order object and an order    *
*   object.                                                    *
*****
        INVOKE OldOrder    "somNew"    RETURNING    oldOrderObj.

*****
*   Invoke the userinterface object with ReadOrder method.    *
*****
        INVOKE userObj "ReadOrder"    RETURNING    ORDER-NUMBER.

*****
*   Invoke the oldorder object to check the status of         *
*   ordered items.                                            *
*****
        INVOKE oldOrderObj "CheckItems" USING    ORDER-NUMBER
                                RETURNING    OUT-ITEMS.

*****
*   Invoke the userinterface object to write the status of    *
*   out-of-stock items.                                       *
*****
        INVOKE userObj "WriteStatus"    USING    OUT-ITEMS.

```

```

*****
*   Free the oldorder object and the order object.           *
*****
      INVOKE oldOrderObj "somFree".

CHECK-EXIT.
EXIT.
EJECT
CREATE-NEW-ORDER.
*****
*   We will simply use the system date for the order date    *
*   and generate a random number for the order number.      *
*   Also we will initialize the item-count field, which will *
*   control how many items are placed in the order.         *
*****
      MOVE FUNCTION CURRENT-DATE TO ORDER-DATE.

      COMPUTE WS-RANDOM-VAL = FUNCTION RANDOM.
      COMPUTE ORDER-NUMBER = WS-RANDOM-VAL * 10000.

      MOVE     ZERO      TO  ITEM-COUNT.

*****
*   Invoke the Order class with the inherited method somNew  *
*   to instantiate an order object.                         *
*****
      INVOKE WineOrder      "somNew"  RETURNING  orderObj.

*****
*   Invoke the setordernumber and setorderdate methods to   *
*   set the order's date and number.                       *
*****
      INVOKE orderObj "SetOrderNumber" USING ORDER-NUMBER.
      INVOKE orderObj "SetOrderDate"   USING ORDER-DATE.

*****
*   Invoke the userinterface object with ReadAction method. *
*****
      INVOKE userObj "ReadAction" RETURNING ACTION.

*****
*   Loop until the user signals the end of the order.       *
*****
      PERFORM UNTIL ACTION (1:3) = "END"
          OR ITEM-COUNT = MAX-ITEMS

          EVALUATE ACTION (1:3)
              WHEN "ADD"
*
*               get the type and cost from the user interface
*               INVOKE userObj "ReadType"  RETURNING ITEM-TYPE
*               INVOKE userObj "ReadCost"  RETURNING ITEM-COST
*
*               instantiate a bottle with those attributes
*               INVOKE Bottle  "somNew"  RETURNING bottleObj
*               INVOKE bottleObj "SetType" USING ITEM-TYPE
*               INVOKE bottleObj "SetCost" USING ITEM-COST
*
*               add it to the collection in the order

```

```

        INVOKE orderObj "AddBottle" USING      bottleObj
        RETURNING WS-PARMS

*           if the add failed, destroy the object just
*           created because we can't do anything with it
*           and it's not in the collection.
*           "0" = success; "1" = failure.
        IF FAILURE
            THEN INVOKE bottleObj  "somFree"
        END-IF

*           send appropriate msg via the user interface
        INVOKE userObj  "WriteMessage" USING WS-FLAG

    WHEN "DEL"

*           get the type and cost from the user interface
        INVOKE userObj "ReadType"  RETURNING ITEM-TYPE
        INVOKE userObj "ReadCost"  RETURNING ITEM-COST

*           create a bottle with those attributes
        INVOKE Bottle  "somNew"    RETURNING bottleObj
        INVOKE bottleObj "SetType"  USING      ITEM-TYPE
        INVOKE bottleObj "SetCost"  USING      ITEM-COST

*           delete copies of it from the collection
        INVOKE orderObj "RemoveBottle" USING bottleObj
        RETURNING
        WS-PARMS

*           destroy the bottle just created
        INVOKE bottleObj  "somFree"

*           send appropriate msg via the user interface
        INVOKE userObj  "WriteMessage" USING WS-FLAG

    WHEN OTHER
        CONTINUE
    END-EVALUATE
    INVOKE userObj "ReadAction"  RETURNING ACTION
    END-PERFORM.
*****
*   End of loop.
*****

*****
*   Close-out processing follows.
*****

*****
*   If no items were ordered, end the process here.
*****
    IF ITEM-COUNT = 0
        *****
        *   Free the order object we created.
        *****
        THEN INVOKE orderObj  "somFree"
        GO TO CREATE-EXIT
    END-IF.

```

```

*****
*   Invoke the order object with the calculate cost method.   *
*****
        INVOKE orderObj "CalculateCost"      RETURNING TOTAL-COST.

*****
*   Invoke the order object with the getordernumber          *
*   and the getorderdate methods.                            *
*****
        INVOKE orderObj "GetOrderNumber"     RETURNING ORDER-NUMBER.
        INVOKE orderObj "GetOrderDate"      RETURNING ORDER-DATE.

*****
*   Invoke the userinterface object with writeoutput method. *
*****
        INVOKE userObj "WriteOutput"         USING TOTAL-COST
                                                ORDER-NUMBER
                                                ORDER-DATE.

*****
*   Invoke the order object with the describeorder method.   *
*****
        INVOKE orderObj "DescribeOrder" RETURNING WS-ITEMS.

*****
*   Invoke the userinterface object with writeoutput method. *
*****
        INVOKE userObj "WriteBottle"        USING WS-ITEMS.

*****
*   Create a filereadwrite object with the inherited somNew   *
*   method.                                                    *
*****
        INVOKE FileRW      "somNew" RETURNING fileObj.

*****
*   Invoke the file object with the xternorder method.        *
*****
        INVOKE fileObj "XternOrder" USING orderObj.

*****
*   Free the filereadwrite object we created.                  *
*****
        INVOKE fileObj "somFree".

*****
*   Free the order object we created.                          *
*****
        INVOKE orderObj "somFree".

CREATE-EXIT.
EXIT.
END PROGRAM "Wine".

```



---

## F.2 Wine Order Class Code

```
process pgmname(mixed) test
  IDENTIFICATION DIVISION.

*****
*   Class WineOrder : Inherits from SOMObject           *
*                                   in the SOM Class Library. *
*****

CLASS-ID.  "WineOrder" INHERITS SOMObject.

*****
*   Class WineOrder contains the following methods:      *
*   somDefaultInit -   Initializes a WineOrder object.   *
*   somFree           -   Frees bottles, collection, and order. *
*   SetOrderNumber    -   Sets the number of a WineOrder object *
*                                   based on a given object reference. *
*   SetOrderDate      -   Sets the date of a WineOrder object *
*                                   based on a given object reference. *
*   AddBottle         -   Adds a bottle object to the order *
*   RemoveBottle      -   Removes a bottle object from the *
*                                   order. *
*   CalculateCost     -   Computes the cost of the bottle *
*                                   objects in the order. *
*   DescribeOrder     -   Lists the contents of the bottles *
*                                   collected in the order. *
*   GetOrderNumber    -   Retrieves the number of a WineOrder *
*                                   object. *
*   GetOrderDate      -   Retrieves the date of a WineOrder *
*                                   object. *
*   SetInstanceData-   Sets all the attributes of an order *
*                                   object. *
*   GetInstanceData-   Gets all the attributes of an order *
*                                   object. *
*   GetEV             -   Retrieves the SOM environment *
*                                   variable. *
*   GetList           -   Retrieves the SOM list for the *
*                                   collected order items. *
*   GetIterator       -   Retrieves the SOM iterator for the *
*                                   collected order items. *
*****

ENVIRONMENT DIVISION.

*****
*   Define which classes will be used by the methods in   *
*   this class. *
*****

CONFIGURATION SECTION.
REPOSITORY.
    CLASS SOMObject          IS "SOMObject"
    CLASS SOMCollection      IS "somf_TSet"
    CLASS SOMIterator        IS "somf_TSetIterator"
    CLASS WineBottle         IS "WineBottle".

*****
*   Define the WineOrder Object. *
*****
```

```

*****

DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-EV          USAGE POINTER.

*****
*   Define the instance data of the WineOrder Object.   *
*****
01 WINEORDER-OBJECT.
   05 WINEORDER-NUMBER      PIC X(5).
   05 WINEORDER-DATE        PIC X(8).
   05 WINEORDER-LIST  USAGE OBJECT REFERENCE SOMCollection.

*****
*   Define an iterator for use on the wineorder-list.   *
*****
01 WINEORDER-ITERATOR  USAGE OBJECT REFERENCE SOMIterator.
   EJECT

PROCEDURE DIVISION.

*****
*****
*   The overridden method somDefaultInit initializes the *
*   WineOrder instance, and creates the collection to be *
*   used in the order.                                   *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "somDefaultInit"      OVERRIDE.

DATA DIVISION.

PROCEDURE DIVISION.

*****
*   Initialize the SOM global environment variable.   *
*****
CALL "somGetGlobalEnvironment" RETURNING WS-EV.
*****
*   Now initialize an empty collection for us to add bottles *
*   into with the addBottle method.                     *
*****
INVOKE SOMCollection "somNew"
      RETURNING WINEORDER-LIST.

*****
*   Instantiate an iterator object and associate it with the *
*   collection.                                             *
*****
INVOKE WINEORDER-LIST "somfCreateIterator"
      USING      BY VALUE WS-EV
      RETURNING WINEORDER-ITERATOR.

*****
*   EXIT and END the method.                               *
*****
EXIT METHOD.
END METHOD "somDefaultInit".

```

```

EJECT
*****
*****
*   The overridden method somFree      destroys the bottle      *
*   objects created, the collection object, and the order      *
*   object.                                                     *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "somFree"          OVERRIDE.

DATA DIVISION.
WORKING-STORAGE SECTION.
01 CollectedBottle          USAGE OBJECT REFERENCE WineBottle.
01 ITEM-COUNT                PIC S9(8)    COMP.

PROCEDURE DIVISION.

*****
*   Get the collected objects.                                     *
*****
        INVOKE WINEORDER-LIST "somfDeleteAll"
                USING          BY VALUE WS-EV.

*****
*   Free the list and iterator objects                             *
*****
        INVOKE WINEORDER-ITERATOR "somFree".

        INVOKE WINEORDER-LIST "somFree".

*****
*   Free thyself...Use SUPER so we don't recurse back into      *
*   this method.                                                 *
*****
        INVOKE SUPER "somFree".

*****
*   EXIT and END the method.                                     *
*****
        EXIT METHOD.
        END METHOD "somFree".
EJECT
*****
*****
*   Method GetOrderNumber gets the number of WineOrder based   *
*   on the object reference of the WineOrder.                 *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "GetOrderNumber".

DATA DIVISION.
WORKING-STORAGE SECTION.

*****
*   Define the linkage attributes.                                 *
*****
LINKAGE SECTION.
01 LS-ORDERNUMBER          PIC X(5).

```



```

*****
MOVE LS-ORDERNUMBER TO WINEORDER-NUMBER.

EXIT METHOD.
END METHOD "SetOrderNumber".
EJECT
*****
*****
*   Method SetOrderDate  sets the date of a WineOrder based  *
*   on the object reference of the WineOrder.                *
*****

IDENTIFICATION DIVISION.
METHOD-ID. "SetOrderDate".

DATA DIVISION.
WORKING-STORAGE SECTION.

*****
*   Define the linkage attributes.                             *
*****

LINKAGE SECTION.
01  LS-ORDERDATE                                PIC X(8).

PROCEDURE DIVISION                                USING      LS-ORDERDATE.

*****
*   Move data from the LINKAGE SECTION.                       *
*****

MOVE LS-ORDERDATE TO WINEORDER-DATE.

EXIT METHOD.
END METHOD "SetOrderDate".
EJECT
*****
*****
*   Method DescribeOrder describes the order contents.        *
*****

IDENTIFICATION DIVISION.
METHOD-ID. "DescribeOrder".

DATA DIVISION.

LOCAL-STORAGE SECTION.
01  CollectedBottle  USAGE OBJECT REFERENCE WineBottle.
01  WS-TYPE                                PIC X(20).
01  WS-COST                                PIC 999V99.
01  ITEM-COUNT                                PIC S9(8)  COMP.

LINKAGE SECTION.
01  LS-ITEMS.
    05  LS-ITEM-COUNT                                PIC S9(4).
    05  LS-ITEM                                OCCURS 1 TO 64 TIMES
                                                DEPENDING ON LS-ITEM-COUNT
                                                INDEXED BY  LS-INDEX.
        10  LS-TYPE                                PIC X(20).
        10  LS-COST                                PIC 999V99.

PROCEDURE DIVISION                                RETURNING LS-ITEMS.

```

```

*****
*   Get the count of the number of items in the collection.   *
*****
        INVOKE WINEORDER-LIST "somfCount"
                USING      BY VALUE WS-EV
                RETURNING   ITEM-COUNT.
        MOVE ITEM-COUNT TO LS-ITEM-COUNT.

*****
*   Get the first one in the collection.                       *
*****
        IF ITEM-COUNT > 0
            THEN SET LS-INDEX TO 1
                INVOKE WINEORDER-ITERATOR "somfFirst"
                        USING      BY VALUE WS-EV
                        RETURNING CollectedBottle
                PERFORM GET-TYPE-N-COST
        END-IF.

*****
*   Get the rest...                                           *
*****
        SUBTRACT 1 FROM ITEM-COUNT.
        IF ITEM-COUNT > 0
            THEN PERFORM ITEM-COUNT TIMES
                SET LS-INDEX UP BY 1
                INVOKE WINEORDER-ITERATOR "somfNext"
                        USING      BY VALUE WS-EV
                        RETURNING CollectedBottle
                PERFORM GET-TYPE-N-COST
            END-PERFORM
        END-IF.

*****
*   Exit and end the method.                                   *
*****
        EXIT METHOD.

*****
*   Invoke the gettype and getcost methods on the bottle     *
*   object and move the returned attributes to the table.     *
*****
        GET-TYPE-N-COST.
            INVOKE CollectedBottle "GetType" RETURNING WS-TYPE.
            MOVE WS-TYPE TO LS-TYPE (LS-INDEX).
            INVOKE CollectedBottle "GetCost" RETURNING WS-COST.
            MOVE WS-COST TO LS-COST (LS-INDEX).

        END METHOD "DescribeOrder".
        EJECT

*****
*   Method CalculateCost computes the cost of the order.      *
*****
        IDENTIFICATION DIVISION.
        METHOD-ID. "CalculateCost".

        DATA DIVISION.
        WORKING-STORAGE SECTION.

```

```

01 CollectedBottle      USAGE OBJECT REFERENCE WineBottle.
01 ITEM-COUNT            PIC S9(8)  COMP.
01 WS-COST               PIC 999V99.

*****
*   Define the linkage attributes.                               *
*****

LINKAGE SECTION.
01 LS-COST               PIC 9(7)V99.

PROCEDURE DIVISION
                                RETURNING  LS-COST.

*****
*   Initialize the accumulator for the total cost.              *
*****

    MOVE ZERO TO LS-COST.

*****
*   Get the count of the number of items in the collection.     *
*****

    INVOKE WINEORDER-LIST "somfCount"
                                USING      BY VALUE WS-EV
                                RETURNING  ITEM-COUNT.

*****
*   Get the first one in the collection.                         *
*****

    IF ITEM-COUNT > 0
        INVOKE WINEORDER-ITERATOR "somfFirst"
                                USING      BY VALUE WS-EV
                                RETURNING  CollectedBottle
        PERFORM GET-COST
    END-IF.

*****
*   Get the rest...                                             *
*****

    SUBTRACT 1 FROM ITEM-COUNT.
    IF ITEM-COUNT > 0
        THEN PERFORM ITEM-COUNT TIMES
            INVOKE WINEORDER-ITERATOR "somfNext"
                                USING      BY VALUE WS-EV
                                RETURNING  CollectedBottle
            PERFORM GET-COST
        END-PERFORM
    END-IF.

*****
*   EXIT the method and return.                                  *
*****

    EXIT METHOD.

*****
*   Invoke the getcost method on the bottle object and          *
*   accumulate the cost.                                         *
*****

    GET-COST.
    INVOKE CollectedBottle "GetCost" RETURNING WS-COST.
    ADD WS-COST TO LS-COST.

```

```

END METHOD "CalculateCost".
EJECT
*****
*****
*   Method AddBottle adds a bottle of wine to the bottle   *
*   collection in the wine order.                           *
*****

IDENTIFICATION DIVISION.
METHOD-ID. "AddBottle".

DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-BEFORE-COUNT          PIC S9(8)    COMP.
01 WS-AFTER-COUNT          PIC S9(8)    COMP.
01 CollectedBottle        USAGE OBJECT REFERENCE WineBottle.

01 theEqualFlag            PIC X.
01 ITEM-FOUND-FLAG         PIC X.
01 ITEM-COUNT              PIC S9(8)    COMP.
01 LOOP-COUNT              PIC S9(8)    COMP.

*****
*   Define the linkage attributes.                           *
*****

LINKAGE SECTION.
01 LS-BOTTLE                USAGE OBJECT REFERENCE WineBottle.
01 LS-PARMS.
   05 LS-ITEM-COUNT          PIC S9(8)    COMP.
   05 LS-FLAG                PIC X.

PROCEDURE DIVISION
                                USING      LS-BOTTLE
                                RETURNING   LS-PARMS.

                                MOVE LOW-VALUE      TO ITEM-FOUND-FLAG.

*****
*   Get the count of items before adding the object.        *
*****
                                INVOKE WINEORDER-LIST "somfCount"
                                                USING      BY VALUE WS-EV
                                                RETURNING WS-BEFORE-COUNT.
                                MOVE      WS-BEFORE-COUNT TO ITEM-COUNT.

*****
*   Get the first one in the collection.                    *
*****
                                IF ITEM-COUNT NOT = 0
                                    THEN INVOKE WINEORDER-ITERATOR "somfFirst"
                                                USING      BY VALUE WS-EV
                                                RETURNING CollectedBottle
                                    PERFORM CHECK-EQUAL
                                END-IF.

*****
*   Get the rest...                                         *
*****
                                SUBTRACT 1 FROM ITEM-COUNT.
                                IF ITEM-COUNT > 0

```



```

        THEN PERFORM VARYING LOOP-COUNT
            FROM 1 BY 1
            UNTIL LOOP-COUNT      > ITEM-COUNT
                OR ITEM-FOUND-FLAG = HIGH-VALUE
            INVOKE WINEORDER-ITERATOR "somfNext"
                USING      BY VALUE WS-EV
                RETURNING CollectedBottle
            PERFORM CHECK-EQUAL
        END-PERFORM
    END-IF.

*****
*   Add the bottle to the collection if it hasn't been   *
*   added before.                                         *
*****
        IF ITEM-FOUND-FLAG = LOW-VALUE
            THEN INVOKE WINEORDER-LIST "somfAdd"
                USING BY VALUE WS-EV
                BY VALUE LS-BOTTLE.

*****
*   Get the count of items after adding the object.       *
*****
        INVOKE WINEORDER-LIST "somfCount"
            USING      BY VALUE WS-EV
            RETURNING WS-AFTER-COUNT.
        MOVE WS-AFTER-COUNT    TO LS-ITEM-COUNT.

*****
*   If the counts are the same the add failed.           *
*****
        IF WS-BEFORE-COUNT = WS-AFTER-COUNT
            THEN MOVE "1" TO LS-FLAG
        ELSE
            MOVE "0" TO LS-FLAG
        END-IF.

*****
*   EXIT the method and return.                           *
*****
        EXIT METHOD.

*****
*   Invoke the somfIsEqual method in the bottle object to *
*   see if the objects are equal. Set a flag if they are. *
*****
        CHECK-EQUAL.
        INVOKE CollectedBottle "somfIsEqual"
            USING      BY VALUE WS-EV
            BY VALUE LS-BOTTLE
            RETURNING theEqualFlag.
        IF theEqualFlag = HIGH-VALUE
            THEN MOVE HIGH-VALUE TO ITEM-FOUND-FLAG.

    END METHOD "AddBottle".
    EJECT
*****
*   Method RemoveBottle removes a bottle from the bottle *

```

```

*      collection in the wine order.                                     *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "RemoveBottle".

DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-BEFORE-COUNT          PIC S9(8)      COMP.
01 WS-AFTER-COUNT          PIC S9(8)      COMP.
01 CollectedBottle        USAGE OBJECT REFERENCE WineBottle.
01 theEqualFlag            PIC X.
01 ITEM-COUNT              PIC S9(8)      COMP.
01 LOOP-COUNT              PIC S9(8)      COMP.

*****
*      Define the linkage attributes.                                     *
*****
LINKAGE SECTION.
01 LS-BOTTLE                USAGE OBJECT REFERENCE WineBottle.
01 LS-PARMS.
   05 LS-ITEM-COUNT          PIC S9(8)      COMP.
   05 LS-FLAG                PIC X.

PROCEDURE DIVISION
                                USING      LS-BOTTLE
                                RETURNING   LS-PARMS.

*****
*      Get the count of items before the delete.                         *
*****
      INVOKE WINEORDER-LIST "somfCount"
                                USING      BY VALUE WS-EV
                                RETURNING   WS-BEFORE-COUNT.
      MOVE WS-BEFORE-COUNT TO ITEM-COUNT.

*****
*      Get the first one in the collection.                               *
*****
      IF ITEM-COUNT NOT = 0
          THEN INVOKE WINEORDER-ITERATOR "somfFirst"
                                USING      BY VALUE WS-EV
                                RETURNING   CollectedBottle
          PERFORM CHECK-EQUAL-N-REMOVE
      END-IF.

*****
*      Get the rest...                                                    *
*****
      SUBTRACT 1 FROM ITEM-COUNT.
      IF ITEM-COUNT > 0
          THEN PERFORM VARYING LOOP-COUNT
                                FROM 1 BY 1
                                UNTIL LOOP-COUNT > ITEM-COUNT
                                OR theEqualFlag = HIGH-VALUE
          INVOKE WINEORDER-ITERATOR "somfNext"
                                USING      BY VALUE WS-EV
                                RETURNING   CollectedBottle
          PERFORM CHECK-EQUAL-N-REMOVE
      END-PERFORM

```

```

END-IF.

*****
*   Get the count of items after the delete.   *
*****
        INVOKE WINEORDER-LIST "somfCount"
                USING      BY VALUE WS-EV
                RETURNING WS-AFTER-COUNT.
        MOVE WS-AFTER-COUNT   TO LS-ITEM-COUNT.

*****
*   If the counts are the same the delete failed.   *
*****
        IF WS-BEFORE-COUNT = WS-AFTER-COUNT
                THEN MOVE "1" TO LS-FLAG
        ELSE
                MOVE "0" TO LS-FLAG
        END-IF.

*****
*   EXIT the method and return.   *
*****
        EXIT METHOD.

CHECK-EQUAL-N-REMOVE.
        INVOKE CollectedBottle "somfIsEqual"
                USING      BY VALUE WS-EV
                BY VALUE LS-BOTTLE
                RETURNING      theEqualFlag.
*****
*   If we find one, remove it from the list.   *
*****
        IF theEqualFlag = HIGH-VALUE
                THEN INVOKE WINEORDER-LIST "somfRemove"
                        USING BY VALUE WS-EV
                        BY VALUE CollectedBottle
                INVOKE CollectedBottle "somFree".

END METHOD "RemoveBottle".
EJECT

*****
*****
*   Method GetEV gets the SOM environment pointer.   *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "GetEV".

DATA DIVISION.
WORKING-STORAGE SECTION.

*****
*   Define the linkage attributes.   *
*****
LINKAGE SECTION.
01 LS-EV                      USAGE POINTER.

PROCEDURE DIVISION
                                RETURNING      LS-EV.
*****

```

```

*      Move data to the LINKAGE SECTION.                                     *
*****
      SET LS-EV TO WS-EV.

      EXIT METHOD.
      END METHOD "GetEV".
      EJECT
*****
*****
*      Method GetList gets the wineorder list collection.                 *
*****

      IDENTIFICATION DIVISION.
      METHOD-ID. "GetList".

      DATA DIVISION.
      WORKING-STORAGE SECTION.

*****
*      Define the linkage attributes.                                       *
*****

      LINKAGE SECTION.
      01 LS-LIST                  USAGE OBJECT REFERENCE SOMCollection.

      PROCEDURE DIVISION          RETURNING    LS-LIST.

*****
*      Move data to the LINKAGE SECTION.                                     *
*****

      SET LS-LIST TO WINEORDER-LIST.

      EXIT METHOD.
      END METHOD "GetList".
      EJECT
*****
*****
*      Method GetIterator gets the wineorder list iterator.               *
*****

      IDENTIFICATION DIVISION.
      METHOD-ID. "GetIterator".

      DATA DIVISION.
      WORKING-STORAGE SECTION.

*****
*      Define the linkage attributes.                                       *
*****

      LINKAGE SECTION.
      01 LS-ITERATOR USAGE OBJECT REFERENCE SOMIterator.

      PROCEDURE DIVISION          RETURNING    LS-ITERATOR.

*****
*      Move data to the LINKAGE SECTION.                                     *
*****

      SET LS-ITERATOR TO WINEORDER-ITERATOR.

      EXIT METHOD.
      END METHOD "GetIterator".
      EJECT

```

```

*****
*****
*   Method SetInstanceData sets all the attributes of an   *
*   order object from data passed in the linkage section.  *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "SetInstanceData".

DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-PARMS.
    05 ITEM-COUNT          PIC S9(8)      COMP.
    05 WS-FLAG             PIC X.
        88 SUCCESSFUL          VALUE "0".
        88 FAILURE            VALUE "1".
01 bottleObj              USAGE OBJECT REFERENCE WineBottle.

*****
*   Define the linkage attributes.                          *
*****
LINKAGE SECTION.
01 LS-ORDER.
    05 LS-ORDER-NUMBER      PIC X(5).
    05 LS-ORDER-DATE        PIC X(8).
    05 FILLER               PIC XXX.
    05 LS-ORDER-COUNT       PIC S9(4).
    05 LS-ORDER-ITEM        OCCURS 1 TO 64 TIMES
                           DEPENDING ON LS-ORDER-COUNT
                           INDEXED BY LS-INDEX.
        10 LSO-TYPE          PIC X(20).
        10 LSO-COST          PIC 999V99.

PROCEDURE DIVISION
                                USING      LS-ORDER.

*****
*   Move in the easy stuff...                               *
*****
    INVOKE self "SetOrderNumber" USING LS-ORDER-NUMBER.
    INVOKE self "SetOrderDate"   USING LS-ORDER-DATE.

*****
*   And now the tricky stuff...                             *
*****
    PERFORM VARYING LS-INDEX FROM 1 BY 1
        UNTIL LS-INDEX > LS-ORDER-COUNT
        INVOKE WineBottle "somNew" RETURNING bottleObj
        INVOKE bottleObj "SetType" USING LSO-TYPE (LS-INDEX)
        INVOKE bottleObj "SetCost" USING LSO-COST (LS-INDEX)
        INVOKE self "AddBottle"   USING bottleObj
                                RETURNING WS-PARMS

    END-PERFORM.

    EXIT METHOD.
END METHOD "SetInstanceData".
EJECT

*****
*   Method GetInstanceData retrieves all the attributes of an *

```

```

*      order object and places them in the linkage section.      *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "GetInstanceData".

DATA DIVISION.
WORKING-STORAGE SECTION.

*****
*      Define the linkage attributes.      *
*****
LINKAGE SECTION.
01 LS-ORDER.
   05 LS-ORDER-NUMBER          PIC X(5).
   05 LS-ORDER-DATE           PIC X(8).
   05 FILLER                   PIC XXX.
   05 LS-ITEMS.
      10 LS-ORDER-COUNT        PIC S9(4).
      10 LS-ORDER-ITEM         OCCURS 1 TO 64 TIMES
                                DEPENDING ON LS-ORDER-COUNT
                                INDEXED BY LS-INDEX.
      15 LSO-TYPE              PIC X(20).
      15 LSO-COST              PIC 999V99.

PROCEDURE DIVISION
                                RETURNING LS-ORDER.

    INVOKE self "GetOrderNumber" RETURNING LS-ORDER-NUMBER.
    INVOKE self "GetOrderDate"   RETURNING LS-ORDER-DATE.
    INVOKE self "DescribeOrder"  RETURNING LS-ITEMS.

    EXIT METHOD.
END METHOD "GetInstanceData".
SKIP3
SKIP3
*****
*      End object definition and class WineOrder.      *
*****
END CLASS "WineOrder".

```

---

### F.3 Old Order Class Code

```

process test pgmname(longmixed)
IDENTIFICATION DIVISION.

*****
*      Class OldOrder : Inherits from the WineOrder class.      *
*****

CLASS-ID. "OldOrder" INHERITS WineOrder.

*****
*      Class OldOrder contains the following methods:      *
*      CheckItems      - Checks the status of ordered items.  *
*      somFree          - Overridden method that invokes      *
*                        destructor in parent class.           *
*****

```

# ENVIRONMENT DIVISION.

```
*****
*   Define which classes will be used by the methods in   *
*   this class.                                           *
*****
```

## CONFIGURATION SECTION.

### REPOSITORY.

```
CLASS OldOrder      IS "OldOrder"
CLASS WineOrder     IS "WineOrder"
CLASS WineBottle    IS "WineBottle"
CLASS FileRW        IS "FileRW"
CLASS SOMCollection IS "somf_TSet"
CLASS SOMIterator   IS "somf_TSetIterator".
```

## DATA DIVISION.

### PROCEDURE DIVISION.

```
*****
*****
*   Method CheckItems checks to see if an item is in stock *
*   or not.                                                 *
*****
```

## IDENTIFICATION DIVISION.

METHOD-ID. "CheckItems".

## DATA DIVISION.

### WORKING-STORAGE SECTION.

```
01 CollectedBottle  USAGE OBJECT REFERENCE WineBottle.
01 WINEORDER-LIST    USAGE OBJECT REFERENCE SOMCollection.
01 WINEORDER-ITERATOR USAGE OBJECT REFERENCE SOMIterator.
01 fileObj           USAGE OBJECT REFERENCE FileRW.
01 WS-FLAG           PIC X.
   88 OUT-OF-STOCK          VALUE "0".
   88 IN-STOCK              VALUE "1".
01 WS-TYPE           PIC X(20).
01 WS-COST           PIC 999V99.
01 WS-EV             USAGE POINTER.
01 ITEM-COUNT        PIC S9(8)  COMP.

01 WS-ORDER-RECORD.
   05 WSO-ORDER-NUMBER PIC X(5).
   05 WSO-ORDER-DATE  PIC X(8).
   05 FILLER          PIC XXX.
   05 WSO-ITEMS.
       10 WSO-ORDER-COUNT PIC S9(4).
       10 WSO-ORDER-ITEM OCCURS 1 TO 64
                           DEPENDING ON WSO-ORDER-COUNT
                           INDEXED BY WSO-INDEX.
           15 WSOR-TYPE PIC X(20).
           15 WSOR-COST PIC 999V99.
```

```
*****
*   Define the linkage attributes.                         *
*****
```

### LINKAGE SECTION.

```
01 LS-ORDER-NUMBER PIC X(5).
01 LS-OUT-ITEMS.
```

```

05 LS-OUT-COUNT          PIC S9(4).
05 LS-OUT-ITEM           OCCURS 1 TO 64
                        DEPENDING ON LS-OUT-COUNT
                        INDEXED BY LS-INDEX.
10 LSO-TYPE              PIC X(20).
10 LSO-COST              PIC 999V99.

PROCEDURE DIVISION
                        USING      LS-ORDER-NUMBER
                        RETURNING  LS-OUT-ITEMS.

*****
*   Create a filereadwrite object with the inherited somNew   *
*   method.                                                  *
*****
      INVOKE FileRW      "somNew"      RETURNING  fileObj.

*****
*   Invoke the file object with the xreadorder method.      *
*****
      INVOKE fileObj     "XReadOrder"    USING  LS-ORDER-NUMBER
                        RETURNING  WS-ORDER-RECORD.

*****
*   Free the filereadwrite object we created.                *
*****
      INVOKE fileObj     "somFree".

*****
*   Set the instance data in the order object with the data  *
*   returned from the file object.                            *
*****
      INVOKE self        "SetInstanceData"  USING WS-ORDER-RECORD.

*****
*   Get the SOM environment variable, and the som collection *
*   from the parent object so we can use them here.          *
*****
      INVOKE self        "GetEV"      RETURNING WS-EV.
      INVOKE self        "GetList"    RETURNING WINEORDER-LIST.
      INVOKE self        "GetIterator" RETURNING WINEORDER-ITERATOR.

*****
*   Get the count of the number of items in the collection.  *
*****
      INVOKE WINEORDER-LIST "somfCount"
                        USING  BY VALUE WS-EV
                        RETURNING  ITEM-COUNT.

*****
*   Get the first one in the collection.                      *
*****
      MOVE ZERO TO LS-OUT-COUNT.
      SET LS-INDEX TO 1.
      IF ITEM-COUNT > 0
          THEN INVOKE WINEORDER-ITERATOR "somfFirst"
                        USING  BY VALUE WS-EV
                        RETURNING CollectedBottle
          PERFORM CHECK-STATUS

```



```

END-IF.
*****
*   Get the rest...                               *
*****
SUBTRACT 1 FROM ITEM-COUNT.
IF ITEM-COUNT > 0
    THEN PERFORM ITEM-COUNT TIMES
        INVOKE WINEORDER-ITERATOR "somfNext"
            USING      BY VALUE WS-EV
            RETURNING CollectedBottle
        PERFORM CHECK-STATUS
    END-PERFORM
END-IF.

EXIT METHOD.

*****
*   Check the inventory status of the bottle by invoking its *
*   GetStatus. If it's out of stock, move it to the table *
*   and increment the out of stock counter.                *
*****
CHECK-STATUS.
    INVOKE CollectedBottle "GetStatus"
        RETURNING WS-FLAG.
IF OUT-OF-STOCK
    THEN ADD 1 TO LS-OUT-COUNT
        INVOKE CollectedBottle "GetType"
            RETURNING WS-TYPE
        MOVE WS-TYPE TO LSO-TYPE (LS-INDEX)
        INVOKE CollectedBottle "GetCost"
            RETURNING WS-COST
        MOVE WS-COST TO LSO-COST (LS-INDEX)
        SET LS-INDEX UP BY 1.

END METHOD "CheckItems".
EJECT
*****
*   The overridden method somFree invokes the destructor *
*   in the parent class.                                *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "somFree"          OVERRIDE.

DATA DIVISION.
WORKING-STORAGE SECTION.

PROCEDURE DIVISION.

    INVOKE SUPER "somFree".

*****
*   EXIT and END the method.                               *
*****
EXIT METHOD.
END METHOD "somFree".
SKIP3
SKIP3
*****

```

```

*      End object definition and class OldOrder.
*
*****
END CLASS "OldOrder".

```

## F.4 User Interface Class Code

```

process pgmname(mixed) test
  IDENTIFICATION DIVISION.

  *****
  *      Class UserInterface: Inherits from SOMObject
  *
  *      in the SOM Class Library.
  *
  *****

  CLASS-ID.  "UserInterface"      INHERITS SOMObject.

  *****
  *      Class UserInterface contains the following methods:
  *
  *      ReadAction      -   Gets the input command from the
  *
  *      system user.
  *
  *      ReadType        -   Gets the type of item from the
  *
  *      system user.
  *
  *      ReadCost        -   Gets the cost of item from the
  *
  *      system user.
  *
  *      WriteMessage    -   Displays a system status message to
  *
  *      the system user.
  *
  *      WriteOutput     -   Displays the cost of the order and
  *
  *      order to the system user.
  *
  *      WriteBottle     -   Displays the attributes of a bottle
  *
  *      collected in the order.
  *
  *      ReadProcess     -   Gets the processing request from the
  *
  *      system user.
  *
  *      ReadOrder       -   Gets the order number from the system
  *
  *      user.
  *
  *      WriteStatus     -   Displays the inventory status of
  *
  *      ordered items to the system user.
  *
  *****

  ENVIRONMENT DIVISION.

  *****
  *      Define which classes will be used by the methods in
  *
  *      this class.
  *
  *****

  CONFIGURATION SECTION.
  REPOSITORY.
    CLASS SOMObject      IS "SOMObject".

  *****
  *      Define the UserInterface Object.
  *
  *****

  DATA DIVISION.
  WORKING-STORAGE SECTION.

  *****

```

```

*      Define the instance data of the UserInterface Object.      *
*****
01  USER-ACTION                      PIC X(10).
    88  UA-ADD                        VALUE "Add".
    88  UA-DELETE                     VALUE "Delete".
    88  UA-END                        VALUE "End".
    88  UA-NEW                        VALUE "New".
    88  UA-STATUS                     VALUE "Status".
    88  UA-EXIT                       VALUE "Exit".

PROCEDURE DIVISION.

*****
*****
*      Method ReadAction gets the system user's command to be      *
*      processed.                                                  *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "ReadAction".

DATA DIVISION.
WORKING-STORAGE SECTION.
01  WS-EDIT-FLAG                      PIC X.

*****
*      Define the linkage attributes.                                *
*****
LINKAGE SECTION.
01  LS-ACTION                      PIC X(10).

PROCEDURE DIVISION    RETURNING    LS-ACTION.

    MOVE LOW-VALUE TO WS-EDIT-FLAG.
    PERFORM UNTIL WS-EDIT-FLAG NOT = LOW-VALUE
        DISPLAY "Enter the action desired: add, delete, end: "
        ACCEPT USER-ACTION FROM SYSIN
        MOVE FUNCTION UPPER-CASE (USER-ACTION) TO USER-ACTION
        MOVE USER-ACTION TO LS-ACTION

        EVALUATE USER-ACTION (1:3)
            WHEN "ADD"
                MOVE HIGH-VALUE TO WS-EDIT-FLAG
            WHEN "DEL"
                MOVE HIGH-VALUE TO WS-EDIT-FLAG
            WHEN "END"
                MOVE HIGH-VALUE TO WS-EDIT-FLAG
            WHEN OTHER
                DISPLAY "Requested action was " USER-ACTION
                DISPLAY "Try again, fumblefingers!!!"
        END-EVALUATE
    END-PERFORM.
    EXIT METHOD.
END METHOD "ReadAction".
EJECT

*****
*****
*      Method ReadType gets the type of item to be processed.      *
*****
IDENTIFICATION DIVISION.

```

METHOD-ID. "ReadType".

DATA DIVISION.

WORKING-STORAGE SECTION.

01 WS-TYPE PIC X(80).

\*\*\*\*\*  
\* Define the linkage attributes. \*

LINKAGE SECTION.

01 LS-TYPE PIC X(20).

PROCEDURE DIVISION RETURNING LS-TYPE.

DISPLAY "Enter the item: ".

ACCEPT WS-TYPE FROM SYSIN.

MOVE FUNCTION UPPER-CASE (WS-TYPE) TO WS-TYPE.

MOVE WS-TYPE (1:20) TO LS-TYPE.

EXIT METHOD.

END METHOD "ReadType".

EJECT

\*\*\*\*\*  
\*\*\*\*\*  
\* Method ReadCost gets the cost of the item to be processed.\*  
\*\*\*\*\*

IDENTIFICATION DIVISION.

METHOD-ID. "ReadCost".

DATA DIVISION.

WORKING-STORAGE SECTION.

01 WS-EDIT-FLAG PIC X.

01 WS-COST-WORK PIC X(6).

\*\*\*\*\*  
\* Define the linkage attributes. \*

LINKAGE SECTION.

01 LS-COST PIC 999V99.

PROCEDURE DIVISION RETURNING LS-COST.

MOVE LOW-VALUE TO WS-EDIT-FLAG.

PERFORM UNTIL WS-EDIT-FLAG = HIGH-VALUE

DISPLAY "Enter the cost: "

ACCEPT WS-COST-WORK FROM SYSIN

COMPUTE LS-COST = FUNCTION NUMVAL (WS-COST-WORK)

IF LS-COST NUMERIC

THEN MOVE HIGH-VALUE TO WS-EDIT-FLAG

ELSE

DISPLAY "Cost is not numeric - try again "

DISPLAY "and get it right this time!!! "

END-IF

END-PERFORM.

EXIT METHOD.

END METHOD "ReadCost".

EJECT

\*\*\*\*\*  
\*\*\*\*\*  
\* Method WriteMessage lets the system user know if the \*

```

*      requested action was successful.                                     *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "WriteMessage".

DATA DIVISION.

*****
*      Define the linkage attributes.                                     *
*****
LINKAGE SECTION.
01  LS-FLAG                                PIC X.

PROCEDURE DIVISION                                USING      LS-FLAG.

    IF LS-FLAG = "0"
        THEN DISPLAY USER-ACTION "successful "
    ELSE
        DISPLAY USER-ACTION "failed "
    END-IF.
    EXIT METHOD.
END METHOD "WriteMessage".
EJECT
*****
*****
*      Method WriteOutput displays the order number and cost          *
*      to the system user.                                           *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "WriteOutput".

DATA DIVISION.
WORKING-STORAGE SECTION.
01  FORMATTED-COST                        PIC $Z,ZZZ,ZZ9.99.

*****
*      Define the linkage attributes.                                     *
*****
LINKAGE SECTION.
01  LS-TOTAL-COST                        PIC 9(7)V99.
01  LS-ORDER-NUMBER                      PIC 9(5).
01  LS-ORDER-DATE                        PIC X(8).

PROCEDURE DIVISION                                USING      LS-TOTAL-COST
                                                    LS-ORDER-NUMBER
                                                    LS-ORDER-DATE.

    MOVE LS-TOTAL-COST TO FORMATTED-COST.
    DISPLAY "Your order number " LS-ORDER-NUMBER
           " placed on "         LS-ORDER-DATE
           " costs "             FORMATTED-COST.
    EXIT METHOD.
END METHOD "WriteOutput".
EJECT
*****
*****
*      Method WriteBottle displays the attributes of bottles         *
*      that have been collected in the order.                         *
*****

```

```
IDENTIFICATION DIVISION.
METHOD-ID. "WriteBottle".
```

```
DATA DIVISION.
```

```
WORKING-STORAGE SECTION.
```

```
01 WS-FORMATTED-COUNT          PIC ZZZ9.
01 WS-FORMATTED-COST           PIC ZZ9.99.
```

```
*****
*   Define the linkage attributes.                                     *
*****
```

```
LINKAGE SECTION.
```

```
01 LS-ITEMS.
   05 LS-ITEM-COUNT             PIC S9(4).
   05 LS-ITEM                   OCCURS 1 TO 64 TIMES
                               DEPENDING ON LS-ITEM-COUNT
                               INDEXED BY LS-INDEX.
                                10 LS-TYPE             PIC X(20).
                                10 LS-COST             PIC 999V99.
```

```
PROCEDURE DIVISION          USING      LS-ITEMS.
```

```
    MOVE LS-ITEM-COUNT TO WS-FORMATTED-COUNT.
    DISPLAY "Contains " WS-FORMATTED-COUNT " items".
    PERFORM VARYING LS-INDEX FROM 1 BY 1
        UNTIL LS-INDEX > LS-ITEM-COUNT
        MOVE LS-COST (LS-INDEX) TO WS-FORMATTED-COST
        DISPLAY LS-TYPE (LS-INDEX) @ " WS-FORMATTED-COST
    END-PERFORM.
    EXIT METHOD.
END METHOD "WriteBottle".
EJECT
```

```
*****
*****
*   Method ReadProcess gets the processing request from the        *
*   system user.                                                    *
*****
```

```
IDENTIFICATION DIVISION.
METHOD-ID. "ReadProcess".
```

```
DATA DIVISION.
```

```
WORKING-STORAGE SECTION.
```

```
01 WS-EDIT-FLAG              PIC X.
```

```
*****
*   Define the linkage attributes.                                     *
*****
```

```
LINKAGE SECTION.
```

```
01 LS-PROCESS                 PIC X(10).
```

```
PROCEDURE DIVISION          RETURNING  LS-PROCESS.
```

```
    MOVE LOW-VALUE TO WS-EDIT-FLAG.
    PERFORM UNTIL WS-EDIT-FLAG NOT = LOW-VALUE
        DISPLAY "Enter process desired: new, status, exit: "
        ACCEPT USER-ACTION FROM SYSIN
        MOVE FUNCTION UPPER-CASE (USER-ACTION) TO USER-ACTION
```

```

MOVE USER-ACTION                                TO LS-PROCESS

EVALUATE USER-ACTION (1:3)
  WHEN "NEW"
    MOVE HIGH-VALUE TO WS-EDIT-FLAG
  WHEN "STA"
    MOVE HIGH-VALUE TO WS-EDIT-FLAG
  WHEN "EXI"
    MOVE HIGH-VALUE TO WS-EDIT-FLAG
  WHEN OTHER
    DISPLAY "Requested process was " USER-ACTION
    DISPLAY "Wrong! Get it right this time!!!"
END-EVALUATE
END-PERFORM.
EXIT METHOD.
END METHOD "ReadProcess".
EJECT
*****
*   Method ReadOrder gets the order number from the system   *
*   user.                                                       *
*****

IDENTIFICATION DIVISION.
METHOD-ID. "ReadOrder".

DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-EDIT-FLAG                PIC X.
01 WS-ORDER                    PIC X(5).
01 WS-ORDER-9                  PIC 9(5).

*****
*   Define the linkage attributes.                               *
*****

LINKAGE SECTION.
01 LS-ORDER                    PIC X(5).

PROCEDURE DIVISION
    RETURNING    LS-ORDER.

    MOVE LOW-VALUE TO WS-EDIT-FLAG.
    PERFORM UNTIL WS-EDIT-FLAG = HIGH-VALUE
        DISPLAY "Enter the order number: "
        ACCEPT WS-ORDER FROM SYSIN
        COMPUTE WS-ORDER-9 = FUNCTION NUMVAL (WS-ORDER)
        MOVE WS-ORDER-9 TO LS-ORDER
        IF LS-ORDER NUMERIC
            THEN MOVE HIGH-VALUE TO WS-EDIT-FLAG
        ELSE
            DISPLAY "Order number is not numeric - try again "
            DISPLAY "and get it right this time!!!"
        END-IF
    END-PERFORM.

    EXIT METHOD.
END METHOD "ReadOrder".
EJECT
*****
*   Method WriteStatus displays the inventory attributes and   *
*****

```

```

*      out-of-stock items that are in the order.      *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "WriteStatus".

DATA DIVISION.

WORKING-STORAGE SECTION.

*****
*      Define the linkage attributes.      *
*****
LINKAGE SECTION.
01  LS-OUT-ITEMS.
    05  LS-OUT-COUNT          PIC S9(4).
    05  LS-OUT-ITEM          OCCURS 1 to 64 TIMES
                             DEPENDING ON LS-OUT-COUNT
                             INDEXED BY  LS-INDEX.
        10  LSO-TYPE          PIC X(20).
        10  LSO-COST          PIC 999V99.

PROCEDURE DIVISION
        USING      LS-OUT-ITEMS.

        IF LS-OUT-COUNT > 0
            THEN DISPLAY "LIST OUT OF STOCK ITEMS: "
                    INVOKE SELF "WriteBottle"
                            USING LS-OUT-ITEMS

        ELSE
            DISPLAY "ALL ITEMS IN STOCK!".
        EXIT METHOD.
    END METHOD "WriteStatus".
    SKIP3
    SKIP3
*****
*      End object definition and class UserInterface.      *
*****
END CLASS "UserInterface".

```

---

## F.5 Bottle Class Code

```

process pgmname(mixed) test
    IDENTIFICATION DIVISION.

*****
*      Class WineBottle : Inherits from somf_MCollectible      *
*      in the SOM Class Library.      *
*****

CLASS-ID.  "WineBottle"  INHERITS somf-MCollectible.

*****
*      Class WineBottle contains the following methods:      *
*      somfIsEqual      -  Provides SOM a method to see if two      *
*                        objects are equivalent.      *
*      SetCost          -  Sets the cost of a WineBottle object      *
*                        based on a given object reference.      *
*      SetType          -  Sets the type of a WineBottle object      *

```





```

PROCEDURE DIVISION          USING BY VALUE LS-EV
                             BY VALUE theBottle
                             RETURNING      theEqualFlag.

```

```

*****
*   Get the type and cost of the bottle object   *
*****
      INVOKE theBottle  "GetType"  RETURNING ITEMTYPE.
      INVOKE theBottle  "GetCost"  RETURNING ITEMCOST.

*****
*   Get those just obtained to the attributes of this   *
*   instance.  If they are equal, set the equality flag   *
*   to a binary 1, else set it to a low-value.         *
*****
      IF (WINE-TYPE = ITEMTYPE) AND
         (WINE-COST = ITEMCOST)
         THEN MOVE HIGH-VALUE TO theEqualFlag
      ELSE
         MOVE LOW-VALUE TO theEqualFlag.

      EXIT METHOD.
      END METHOD "somfIsEqual".
      EJECT
*****
*   Method GetType Gets the type of a WineBottle based on the *
*   object reference of the WineBottle.                   *
*****
      IDENTIFICATION DIVISION.
      METHOD-ID. "GetType".

```

```

DATA DIVISION.
WORKING-STORAGE SECTION.

```

```

*****
*   Define the linkage attributes.                       *
*****
      LINKAGE SECTION.
      01 LS-TYPE                                PIC X(20).

      PROCEDURE DIVISION          RETURNING      LS-TYPE.

*****
*   Move data to the LINKAGE SECTION.                   *
*****
      MOVE WINE-TYPE TO LS-TYPE.

      EXIT METHOD.
      END METHOD "GetType".
      EJECT
*****
*   Method GetCost Gets the COST of a WineBottle based on the *
*   object reference of the WineBottle.                   *
*****
      IDENTIFICATION DIVISION.
      METHOD-ID. "GetCost".

```

DATA DIVISION.  
WORKING-STORAGE SECTION.

```
*****
*   Define the linkage attributes.                               *
*****
```

LINKAGE SECTION.

01 LS-COST PIC 999V99.

PROCEDURE DIVISION RETURNING LS-COST.

```
*****
*   Move data to the LINKAGE SECTION.                           *
*****
```

MOVE WINE-COST TO LS-COST.

EXIT METHOD.  
END METHOD "GetCost".  
EJECT

```
*****
*****
*   Method SetType Sets the type of a WineBottle based on the *
*   object reference of the WineBottle.                       *
*****
```

IDENTIFICATION DIVISION.  
METHOD-ID. "SetType".

DATA DIVISION.  
WORKING-STORAGE SECTION.

```
*****
*   Define the linkage attributes.                               *
*****
```

LINKAGE SECTION.

01 LS-TYPE PIC X(20).

PROCEDURE DIVISION USING LS-TYPE.

```
*****
*   Move data to the LINKAGE SECTION.                           *
*****
```

MOVE LS-TYPE TO WINE-TYPE.

EXIT METHOD.  
END METHOD "SetType".  
EJECT

```
*****
*****
*   Method SetCost Sets the COST of a WineBottle based on the *
*   object reference of the WineBottle.                       *
*****
```

IDENTIFICATION DIVISION.  
METHOD-ID. "SetCost".

DATA DIVISION.  
WORKING-STORAGE SECTION.

```
*****
```

```

*      Define the linkage attributes.                                     *
*****
LINKAGE SECTION.
01  LS-COST                               PIC 999V99.

PROCEDURE DIVISION                                USING      LS-COST.

*****
*      Move data to the LINKAGE SECTION.                               *
*****
      MOVE LS-COST TO WINE-COST.

      EXIT METHOD.
      END METHOD "SetCost".
      EJECT
*****
*      Method GetStatus gets the inventory status of an item          *
*      that was ordered.                                                *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "GetStatus".

DATA DIVISION.
WORKING-STORAGE SECTION.
01  WS-STATUS-WORK                        PIC 9(5).
01  WS-STATUS-MOD                        PIC 9.
01  WS-RANDOM-WORK                       PIC 9V9(5).

*****
*      Define the linkage attributes.                                     *
*****
LINKAGE SECTION.
01  LS-STATUS                            PIC X.

PROCEDURE DIVISION                                RETURNING  LS-STATUS.

*****
*      We aren't reading the inventory quantity from an               *
*      external file, so we need to generate our order status         *
*      in here. For starters, we'll assume that we have a 50%         *
*      chance of the item being in stock.                               *
*****
      COMPUTE WS-RANDOM-WORK = FUNCTION RANDOM.
      COMPUTE WS-STATUS-WORK = WS-RANDOM-WORK * 10000.
      DIVIDE WS-STATUS-WORK BY 2 GIVING WS-STATUS-WORK
          REMAINDER WS-STATUS-MOD.
*****
*      If the generated number is even, set the status flag to        *
*      0 which means out-of-stock; else set it to 1, or in-stock.*
*****
      IF WS-STATUS-MOD = 0
          THEN MOVE "0" TO LS-STATUS
      ELSE
          MOVE "1" TO LS-STATUS.

*****
*      Move data to the LINKAGE SECTION.                               *
*****

```

```

        EXIT METHOD.
    END METHOD "GetStatus".
    SKIP3
    SKIP3
    *****
*   End object definition and class WineBottle.   *
    *****
    END CLASS "WineBottle".

```

---

## F.6 FileRW Class Code

```

process pgmname(mixed) test
    IDENTIFICATION DIVISION.

    *****
*   Class FileRW      : Inherits from SOMObject      *
*   in the SOM Class Library.                          *
    *****

    CLASS-ID.  "FileRW"    INHERITS SOMObject.

    *****
*   Class FileRW contains the following methods:      *
*   XternOrder      -   Externalizes an order to a flat  *
*   file.                                                    *
*   XReadOrder      -   Reads the order record from a flat  *
*   file and returns it to the invoker.                  *
    *****

    ENVIRONMENT DIVISION.

    *****
*   Define which classes will be used by the methods in  *
*   this class.                                          *
    *****

    CONFIGURATION SECTION.
    REPOSITORY.
        CLASS SOMObject          IS "SOMObject"
        CLASS WineOrder          IS "WineOrder".

    *****
*   Define the WineOrder Object.                        *
    *****

    DATA DIVISION.
    WORKING-STORAGE SECTION.

    PROCEDURE DIVISION.

    *****
*   Method XternOrder writes the order to a flat file.    *
    *****

    IDENTIFICATION DIVISION.
    METHOD-ID. "XternOrder".

```



```

*      Method XReadOrder reads the order from a flat file.      *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "XReadOrder".

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT ORDERS          ASSIGN TO    ORDERS
    FILE STATUS IS WS-STATUS-FLAG
    ORGANIZATION IS LINE SEQUENTIAL.

DATA DIVISION.
FILE SECTION.
FD  ORDERS EXTERNAL
    RECORD CONTAINS 255.
01  ORDER-RECORD          PIC X(255).

WORKING-STORAGE SECTION.
01  WS-STATUS-FLAG        PIC XX.
01  WS-EOF-FLAG           PIC X.

LINKAGE SECTION.
01  LS-ORDER              PIC X(5).
01  LS-ORDER-RECORD.
    05  LS-ORDER-NUMBER    PIC X(5).
    05  LS-ORDER-DATE      PIC X(8).
    05  FILLER             PIC XXX.
    05  LS-ORDER-COUNT     PIC S9(4).
    05  LS-ORDER-ITEM      OCCURS 1 TO 64
                           DEPENDING ON LS-ORDER-COUNT
                           INDEXED BY LS-INDEX.
    10  LSO-TYPE           PIC X(20).
    10  LSO-COST           PIC 999V99.

PROCEDURE DIVISION    USING LS-ORDER
                      RETURNING LS-ORDER-RECORD.

*****
*      Open the flat file for input; initialize eof flag.      *
*****
OPEN INPUT  ORDERS.
MOVE LOW-VALUE TO WS-EOF-FLAG.

*****
*      Read until the requested order is found on the file.    *
*****
PERFORM UNTIL WS-EOF-FLAG = HIGH-VALUE
    OR LS-ORDER-NUMBER = LS-ORDER
    READ ORDERS INTO LS-ORDER-RECORD
    AT END MOVE HIGH-VALUE TO WS-EOF-FLAG
    NOT AT END
        IF LS-ORDER-NUMBER = LS-ORDER
            THEN CONTINUE
        END-IF
    END-READ
END-PERFORM.

```

```

*****
*   Close the order file after reading the record.   *
*****
      CLOSE ORDERS.

      EXIT METHOD.
END METHOD "XReadOrder".
      SKIP3
      SKIP3
*****
*   End object definition and class FileRW.   *
*****
      END CLASS "FileRW".

```



---

## Appendix G. Wine Store Scenario – Iteration Four Code

This appendix lists all the source modules for the fourth iteration of the Wine Store Scenario.

---

### G.1 Wine Client Code

```
process pgmname(longmixed) test
  IDENTIFICATION DIVISION.
  PROGRAM-ID. "Wine".

*****
*
*   The client program of the wine application does the
*   following tasks:
*   - Instantiates the UserInterface obejct.
*   - If a status request:
*       -- instantiates an OldOrder object
*       -- invokes the UserInterface object to get the order
*         number
*       -- invokes the metaclass to report out-of-stock items
*       -- invokes the UserInterface object to display the
*         status of the out-of-stock items
*       -- invokes the metaclass object to get the count of
*         old orders checked
*       -- invokes the UserInterface object to display the
*         number of orders checked.
*       -- Frees the metaclass object
*   - If a request for a new order:
*       -- Tells the order object to process the user's request
*         and tells the userinterface object to get another
*         request until the user signals the end of the order.
*         If the request is an add or delete, sends the
*         appropriate message to the userinterface object for
*         the item cost and type, as required by the user's
*         processing request.
*       -- Tells the order object to compute the order cost.
*       -- Tells the order object to get the order number.
*       -- Tells the user interface object to write order cost.
*       -- Tells the order to describe itself.
*       -- Tells the order to write itself to the order file.
*       -- Frees the objects it instantiated.
*   - Frees the UserInterface object.
*   - Terminates.
*
*****

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
  CLASS SOMObject           IS "SOMObject"
  CLASS WineOrder           IS "WineOrder"
  CLASS OldOrder            IS "OldOrder"
  CLASS Bottle              IS "WineBottle"
  CLASS FileRW              IS "FileRW"
```

```

CLASS UserInterface          IS "UserInterface".

DATA DIVISION.
WORKING-STORAGE SECTION.

*   OBJECTS:
01 orderObj          USAGE OBJECT REFERENCE WineOrder.
01 oldOrderObj       USAGE OBJECT REFERENCE OldOrder.
01 userObj           USAGE OBJECT REFERENCE UserInterface.
01 bottleObj         USAGE OBJECT REFERENCE Bottle.
01 fileObj           USAGE OBJECT REFERENCE FileRW.
01 metaObj           USAGE OBJECT REFERENCE METAClass OldOrder.

*   DATA ITEMS:
01 ACTION              PIC X(10).
01 PROCESS              PIC X(10).
01 STATUS-FLAG         PIC X.
   88 NO-STATUS-SELECTED      VALUE LOW-VALUES.
   88 STATUS-SELECTED        VALUE HIGH-VALUES.
01 ITEM-TYPE           PIC X(20).
01 ITEM-COST           PIC 999V99.
01 MAX-ITEMS           PIC 9(4)          COMP VALUE 64.
01 WS-PARMS.
   05 ITEM-COUNT         PIC S9(8)          COMP.
   05 WS-FLAG            PIC X.
       88 SUCCESSFUL      VALUE "0".
       88 FAILURE         VALUE "1".

01 ORDER-DATE          PIC X(8).

01 WS-RANDOM-VAL        PIC 9V9(5).
01 ORDER-NUMBER         PIC 9(5).
01 TOTAL-COST           PIC 9(7)V99.
01 WS-ITEMS.
   05 WS-COUNT           PIC S9(4).
   05 WS-ITEM            OCCURS 1 TO 64 TIMES
                        DEPENDING ON WS-COUNT
                        INDEXED BY WS-INDEX.
       10 WS-TYPE        PIC X(20).
       10 WS-COST        PIC 999V99.
01 OUT-ORDERS           PIC S9(4)          COMP.
01 META-PARMS.
   05 univObj           USAGE OBJECT REFERENCE.
   05 LOST-FLAG         PIC X.
   05 OUT-ITEMS.
       10 OUT-COUNT      PIC S9(4).
       10 OUT-ITEM       OCCURS 1 TO 64 TIMES
                        DEPENDING ON OUT-COUNT
                        INDEXED BY OUT-INDEX.
           15 OUT-TYPE    PIC X(20).
           15 OUT-COST    PIC 999V99.

01 WS-ORDER-RECORD.
   05 WSO-ORDER-NUMBER   PIC X(5).
   05 WSO-ORDER-DATE     PIC X(8).
   05 FILLER             PIC XXX.
   05 WSO-ITEMS.
       10 WSO-ORDER-COUNT PIC S9(4).
       10 WSO-ORDER-ITEM OCCURS 1 TO 64

```

```

                                DEPENDING ON WSO-ORDER-COUNT
                                INDEXED BY WSO-INDEX.
                                15 WSO-TYPE PIC X(20).
                                15 WSO-COST PIC 999V99.
EJECT

PROCEDURE DIVISION.

*****
*   Initialized the status selected control flag for use   *
*   in later processing.                                   *
*****
MOVE LOW-VALUES TO STATUS-FLAG.

*****
*   Invoke the UserInterface class with the inherited somNew *
*   method to instantiate a userinterface object.          *
*   somNew is inherited from SOMObject.                    *
*****
INVOKE UserInterface "somNew" RETURNING userObj.

*****
*   Invoke the UserInterface class with the ReadProcess    *
*   method to obtain the process desired by the system user. *
*****
INVOKE userObj "ReadProcess" RETURNING PROCESS.

*****
*   Use the process to control the path through this program. *
*****
PERFORM UNTIL PROCESS (1:4) = "EXIT"
    EVALUATE PROCESS (1:3)
        WHEN "STA"
            PERFORM CHECK-OLD-ORDER THRU CHECK-EXIT
            MOVE HIGH-VALUE TO STATUS-FLAG
        WHEN "NEW"
            PERFORM CREATE-NEW-ORDER THRU CREATE-EXIT
        WHEN OTHER
            CONTINUE
    END-EVALUATE
    INVOKE userObj "ReadProcess" RETURNING PROCESS
END-PERFORM.

*****
*   Check the status flag to see if we need to invoke      *
*   the countoldorders method in the metaclass.           *
*****
IF STATUS-SELECTED
    THEN PERFORM GET-COUNT THRU GET-EXIT.

*****
*   Free the user interface object we created.             *
*****
INVOKE userObj "somFree".

*****
*   We're outta here...                                    *
*****
GOBACK.

```

```

      EJECT
CHECK-OLD-ORDER.
*****
*   Invoke the userinterface object with ReadOrder method.   *
*****
      INVOKE userObj "ReadOrder"      RETURNING  ORDER-NUMBER.

*****
*   Invoke the oldorder object to check the status of         *
*   ordered items.                                           *
*****
      INVOKE OldOrder "CreateOldOrder" USING      ORDER-NUMBER
      RETURNING  META-PARMS.

*****
*   Check to see if the order was found on the file;         *
*   send an error message if it wasn't.                     *
*****
      IF LOST-FLAG = HIGH-VALUE
      THEN INVOKE userObj "WriteLost" USING ORDER-NUMBER
      GO TO CHECK-EXIT.

*****
*   Invoke the userinterface object to write the status of   *
*   out-of-stock items.                                     *
*****
      INVOKE userObj "WriteStatus"  USING      OUT-ITEMS.

CHECK-EXIT.
EXIT.
EJECT
GET-COUNT.

*****
*   Invoke the somGetClass method to get the handle of the   *
*   metaclass object.                                       *
*****
      INVOKE univObj "somGetClass"      RETURNING metaObj.

*****
*   Invoke the metaclass object to get the number of old     *
*   orders.                                                  *
*****
      INVOKE metaObj "CountOldOrders"  RETURNING OUT-ORDERS.

*****
*   Invoke the userinterface object to write out the number  *
*   of old orders.                                           *
*****
      INVOKE userObj "WriteOutCount"  USING OUT-ORDERS.

*****
*   Free the metaclass object.                               *
*****
      INVOKE metaObj "somFree".

GET-EXIT.
EXIT.

```

```

EJECT
CREATE-NEW-ORDER.
*****
*   We will simply use the system date for the order date      *
*   and generate a random number for the order number.        *
*   Also we will initialize the item-count field, which will   *
*   control how many items are placed in the order.           *
*****
MOVE FUNCTION CURRENT-DATE TO ORDER-DATE.

COMPUTE WS-RANDOM-VAL = FUNCTION RANDOM.
COMPUTE ORDER-NUMBER = WS-RANDOM-VAL * 10000.

MOVE     ZERO     TO     ITEM-COUNT.

*****
*   Invoke the Order class with the inherited method somNew    *
*   to instantiate an order object.                            *
*****
INVOKE WineOrder      "somNew"  RETURNING  orderObj.

*****
*   Invoke the setordernumber and setorderdate methods to      *
*   set the order's date and number.                            *
*****
INVOKE orderObj "SetOrderNumber" USING ORDER-NUMBER.
INVOKE orderObj "SetOrderDate"   USING ORDER-DATE.

*****
*   Invoke the userinterface object with ReadAction method.    *
*****
INVOKE userObj "ReadAction" RETURNING ACTION.

*****
*   Loop until the user signals the end of the order.          *
*****
PERFORM UNTIL ACTION (1:3) = "END"
OR ITEM-COUNT = MAX-ITEMS

    EVALUATE ACTION (1:3)
    WHEN "ADD"

        *
        *   get the type and cost from the user interface
        INVOKE userObj "ReadType"  RETURNING ITEM-TYPE
        INVOKE userObj "ReadCost"  RETURNING ITEM-COST

        *
        *   instantiate a bottle with those attributes
        INVOKE Bottle      "somNew"  RETURNING bottleObj
        INVOKE bottleObj "SetType"   USING ITEM-TYPE
        INVOKE bottleObj "SetCost"   USING ITEM-COST

        *
        *   add it to the collection in the order
        INVOKE orderObj "AddBottle" USING bottleObj
        RETURNING WS-PARMS

        *
        *   if the add failed, destroy the object just
        *   created because we can't do anything with it
        *   and it's not in the collection.
        *   "0" = success; "1" = failure.
        IF FAILURE

```

```

        THEN INVOKE bottleObj  "somFree"
    END-IF

*       send appropriate msg via the user interface
    INVOKE userObj  "WriteMessage" USING WS-FLAG

    WHEN "DEL"

*       get the type and cost from the user interface
    INVOKE userObj "ReadType"  RETURNING ITEM-TYPE
    INVOKE userObj "ReadCost"  RETURNING ITEM-COST

*       create a bottle with those attributes
    INVOKE Bottle  "somNew"  RETURNING bottleObj
    INVOKE bottleObj "SetType" USING  ITEM-TYPE
    INVOKE bottleObj "SetCost" USING  ITEM-COST

*       delete copies of it from the collection
    INVOKE orderObj "RemoveBottle" USING bottleObj
                                         RETURNING
                                         WS-PARMS

*       destroy the bottle just created
    INVOKE bottleObj  "somFree"

*       send appropriate msg via the user interface
    INVOKE userObj  "WriteMessage" USING WS-FLAG

    WHEN OTHER
        CONTINUE
    END-EVALUATE
    INVOKE userObj "ReadAction"  RETURNING ACTION
    END-PERFORM.
*****
*       End of loop.
*****

*****
*       Close-out processing follows.
*****

*****
*       If no items were ordered, end the process here.
*****
    IF ITEM-COUNT = 0
*****
*       Free the order object we created.
*****
        THEN INVOKE orderObj  "somFree"
        GO TO CREATE-EXIT
    END-IF.

*****
*       Invoke the order object with the calculate cost method.
*****
    INVOKE orderObj "CalculateCost"  RETURNING TOTAL-COST.

*****
*       Invoke the order object with the getordernumber
*****

```

```

*      and the getorderdate methods.      *
*****
      INVOKE orderObj "GetOrderNumber"      RETURNING ORDER-NUMBER.
      INVOKE orderObj "GetOrderDate"      RETURNING ORDER-DATE.

*****
*      Invoke the userinterface object with writeoutput method. *
*****
      INVOKE userObj "WriteOutput"      USING TOTAL-COST
                                         ORDER-NUMBER
                                         ORDER-DATE.

*****
*      Invoke the order object with the describeorder method. *
*****
      INVOKE orderObj "DescribeOrder" RETURNING WS-ITEMS.

*****
*      Invoke the userinterface object with writeoutput method. *
*****
      INVOKE userObj "WriteBottle"      USING WS-ITEMS.

*****
*      Create a filereadwrite object with the inherited somNew *
*      method.      *
*****
      INVOKE FileRW      "somNew" RETURNING fileObj.

*****
*      Invoke the file object with the xternorder method. *
*****
      INVOKE fileObj "XternOrder"      USING      orderObj.

*****
*      Free the filereadwrite object we created.      *
*****
      INVOKE fileObj "somFree".

*****
*      Free the order object we created.      *
*****
      INVOKE orderObj "somFree".

CREATE-EXIT.
EXIT.
END PROGRAM "Wine".

```

---

## G.2 Old Order Class Code

```

process test pgmname(longmixed)
  IDENTIFICATION DIVISION.

*****
*      Class OldOrder : Inherits from WineOrder, and uses      *
*      the metaclass MetaOldOrder.      *
*****

```

```

CLASS-ID.    "OldOrder"      INHERITS WineOrder
                                   METAClass MetaOldOrder.

```

```

*****
*   Class OldOrder contains the following methods:      *
*   CheckItems      -   Checks the status of ordered items. *
*   somFree         -   Overridden method that invokes   *
*                               destructor in parent class. *
*****

```

ENVIRONMENT DIVISION.

```

*****
*   Define which classes will be used by the methods in  *
*   this class.                                          *
*****

```

CONFIGURATION SECTION.  
REPOSITORY.

```

CLASS OldOrder      IS "OldOrder"
CLASS MetaOldOrder  IS "MetaOldOrder"
CLASS WineOrder     IS "WineOrder"
CLASS WineBottle    IS "WineBottle"
CLASS FileRW        IS "FileRW"
CLASS SOMCollection IS "somf_TSet"
CLASS SOMIterator   IS "somf_TSetIterator".

```

DATA DIVISION.

PROCEDURE DIVISION.

```

*****
*****
*   Method CheckItems checks to see if an item is in stock *
*   or not.                                                  *
*****

```

IDENTIFICATION DIVISION.  
METHOD-ID. "CheckItems".

DATA DIVISION.

WORKING-STORAGE SECTION.

```

01 CollectedBottle  USAGE OBJECT REFERENCE WineBottle.
01 WINEORDER-LIST    USAGE OBJECT REFERENCE SOMCollection.
01 WINEORDER-ITERATOR USAGE OBJECT REFERENCE SOMIterator.
01 fileObj           USAGE OBJECT REFERENCE FileRW.
01 WS-FLAG           PIC X.
   88 OUT-OF-STOCK          VALUE "0".
   88 IN-STOCK              VALUE "1".
01 WS-TYPE           PIC X(20).
01 WS-COST           PIC 999V99.
01 WS-EV            USAGE POINTER.
01 ITEM-COUNT        PIC S9(8)  COMP.

01 WS-ORDER-RECORD.
   05 WSO-ORDER-NUMBER PIC X(5).
   05 WSO-ORDER-DATE  PIC X(8).
   05 FILLER          PIC XXX.
   05 WSO-ITEMS.
       10 WSO-ORDER-COUNT PIC S9(4).
       10 WSO-ORDER-ITEM OCCURS 1 TO 64

```



```

                                DEPENDING ON WSO-ORDER-COUNT
                                INDEXED BY WSO-INDEX.
15 WSOR-TYPE PIC X(20).
15 WSOR-COST PIC 999V99.

*****
*   Define the linkage attributes.   *
*****

LINKAGE SECTION.
01 LS-ORDER-NUMBER PIC X(5).
01 LS-PARMS.
    05 LS-LOST-FLAG PIC X.
    05 LS-OUT-ITEMS.
        10 LS-OUT-COUNT PIC S9(4).
        10 LS-OUT-ITEM OCCURS 1 TO 64
                        DEPENDING ON LS-OUT-COUNT
                        INDEXED BY LS-INDEX.
15 LSO-TYPE PIC X(20).
15 LSO-COST PIC 999V99.

PROCEDURE DIVISION
                                USING LS-ORDER-NUMBER
                                RETURNING LS-PARMS.

*****
*   Create a filereadwrite object with the inherited somNew   *
*   method.   *
*****
                                INVOKE FileRW "somNew" RETURNING fileObj.

*****
*   Invoke the file object with the xreadorder method.   *
*****
                                INVOKE fileObj "XReadOrder" USING LS-ORDER-NUMBER
                                                                RETURNING WS-ORDER-RECORD.

*****
*   Check to see if the order was found on the file, and   *
*   exit with an error if it isn't.   *
*****
                                IF LS-ORDER-NUMBER NOT = WSO-ORDER-NUMBER
                                    THEN MOVE HIGH-VALUE TO LS-LOST-FLAG
                                    EXIT METHOD
                                ELSE
                                    MOVE LOW-VALUE TO LS-LOST-FLAG.

*****
*   Free the filereadwrite object we created.   *
*****
                                INVOKE fileObj "somFree".

*****
*   Set the instance data in the order object with the data   *
*   returned from the file object.   *
*****
                                INVOKE self "SetInstanceData" USING WS-ORDER-RECORD.

*****
*   Get the SOM environment variable, and the som collection   *
*   from the parent object so we can use them here.   *
*****

```

```

*****
        INVOKE self      "GetEV"      RETURNING WS-EV.
        INVOKE self      "GetList"    RETURNING WINEORDER-LIST.
        INVOKE self      "GetIterator"
                                RETURNING WINEORDER-ITERATOR.

*****
*   Get the count of the number of items in the collection.   *
*****
        INVOKE WINEORDER-LIST "somfCount"
                                USING      BY VALUE WS-EV
                                RETURNING   ITEM-COUNT.

*****
*   Get the first one in the collection.                       *
*****
        MOVE ZERO TO LS-OUT-COUNT.
        SET  LS-INDEX TO 1.
        IF  ITEM-COUNT > 0
            THEN INVOKE WINEORDER-ITERATOR "somfFirst"
                                USING      BY VALUE WS-EV
                                RETURNING CollectedBottle
                PERFORM CHECK-STATUS
        END-IF.

*****
*   Get the rest...                                           *
*****
        SUBTRACT 1 FROM ITEM-COUNT.
        IF  ITEM-COUNT > 0
            THEN PERFORM ITEM-COUNT TIMES
                INVOKE WINEORDER-ITERATOR "somfNext"
                                USING      BY VALUE WS-EV
                                RETURNING CollectedBottle
                PERFORM CHECK-STATUS
            END-PERFORM
        END-IF.

        EXIT METHOD.

*****
*   Check the inventory status of the bottle by invoking its  *
*   GetStatus. If it's out of stock, move it to the table    *
*   and increment the out of stock counter.                  *
*****
        CHECK-STATUS.
        INVOKE CollectedBottle "GetStatus"
                                RETURNING WS-FLAG.
        IF  OUT-OF-STOCK
            THEN ADD 1 TO LS-OUT-COUNT
                INVOKE CollectedBottle "GetType"
                                RETURNING WS-TYPE
                MOVE WS-TYPE TO LSO-TYPE (LS-INDEX)
                INVOKE CollectedBottle "GetCost"
                                RETURNING WS-COST
                MOVE WS-COST TO LSO-COST (LS-INDEX)
                SET  LS-INDEX UP BY 1.

        END METHOD "CheckItems".
        EJECT

```

```

*****
*****
*   The overridden method somFree invokes the destructor   *
*   in the parent class.                                   *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "somFree"          OVERRIDE.

DATA DIVISION.
WORKING-STORAGE SECTION.

PROCEDURE DIVISION.

    INVOKE SUPER "somFree".

*****
*   EXIT and END the method.                               *
*****
EXIT METHOD.
END METHOD "somFree".
SKIP3
SKIP3
*****
*   End object definition and class OldOrder.             *
*****
END CLASS "OldOrder".

```

---

### G.3 User Interface Class Code

```

process pgmname(mixed) test
    IDENTIFICATION DIVISION.

*****
*   Class UserInterface: Inherits from SOMObject          *
*   in the SOM Class Library.                             *
*****

CLASS-ID. "UserInterface"    INHERITS SOMObject.

*****
*   Class UserInterface contains the following methods:   *
*   ReadAction      -   Gets the input command from the   *
*   system user.                                          *
*   ReadType        -   Gets the type of item from the    *
*   system user.                                          *
*   ReadCost        -   Gets the cost of item from the    *
*   system user.                                          *
*   WriteMessage    -   Displays a system status message *
*   to the system user.                                  *
*   WriteOutput     -   Displays the cost of the order and *
*   order to the system user.                             *
*   WriteBottle     -   Displays the attributes of a bottle *
*   collected in the order.                               *
*   ReadProcess     -   Gets the processing request from the *
*   system user.                                          *
*   ReadOrder       -   Gets the order number from the system *
*   user.                                                  *
*   WriteStatus     -   Displays the inventory status of   *

```

```

*                ordered items to the system user.      *
*   WriteOutCount - Displays the number of old orders   *
*                checked to the system user.           *
*   WriteLost     - Indicates the order was not found   *
*                on the order file.                    *
*****

```

#### ENVIRONMENT DIVISION.

```

*****
*   Define which classes will be used by the methods in *
*   this class.                                         *
*****

```

#### CONFIGURATION SECTION.

##### REPOSITORY.

```

    CLASS SOMObject          IS "SOMObject".

```

```

*****
*   Define the UserInterface Object.                    *
*****

```

#### DATA DIVISION.

##### WORKING-STORAGE SECTION.

```

*****
*   Define the instance data of the UserInterface Object. *
*****
01 USER-ACTION          PIC X(10).
   88 UA-ADD              VALUE "Add".
   88 UA-DELETE           VALUE "Delete".
   88 UA-END              VALUE "End".
   88 UA-NEW              VALUE "New".
   88 UA-STATUS           VALUE "Status".
   88 UA-EXIT             VALUE "Exit".

```

#### PROCEDURE DIVISION.

```

*****
*****
*   Method ReadAction gets the system user's command to be *
*   processed.                                             *
*****

```

#### IDENTIFICATION DIVISION.

```

METHOD-ID. "ReadAction".

```

#### DATA DIVISION.

##### WORKING-STORAGE SECTION.

```

01 WS-EDIT-FLAG          PIC X.

```

```

*****
*   Define the linkage attributes.                        *
*****

```

#### LINKAGE SECTION.

```

01 LS-ACTION              PIC X(10).

```

#### PROCEDURE DIVISION

```

    RETURNING LS-ACTION.

```

```

MOVE LOW-VALUE TO WS-EDIT-FLAG.
PERFORM UNTIL WS-EDIT-FLAG NOT = LOW-VALUE
    DISPLAY "Enter the action desired: add, delete, end: "
    ACCEPT USER-ACTION FROM SYSIN
    MOVE FUNCTION UPPER-CASE (USER-ACTION) TO USER-ACTION
    MOVE USER-ACTION TO LS-ACTION

    EVALUATE USER-ACTION (1:3)
        WHEN "ADD"
            MOVE HIGH-VALUE TO WS-EDIT-FLAG
        WHEN "DEL"
            MOVE HIGH-VALUE TO WS-EDIT-FLAG
        WHEN "END"
            MOVE HIGH-VALUE TO WS-EDIT-FLAG
        WHEN OTHER
            DISPLAY "Requested action was " USER-ACTION
            DISPLAY "Try again, fumblefingers!!!"
    END-EVALUATE
END-PERFORM.
EXIT METHOD.
END METHOD "ReadAction".
EJECT
*****
*****
* Method ReadType gets the type of item to be processed. *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "ReadType".

DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-TYPE PIC X(80).

*****
* Define the linkage attributes. *
*****
LINKAGE SECTION.
01 LS-TYPE PIC X(20).

PROCEDURE DIVISION
RETURNING LS-TYPE.

    DISPLAY "Enter the item: ".
    ACCEPT WS-TYPE FROM SYSIN.
    MOVE FUNCTION UPPER-CASE (WS-TYPE) TO WS-TYPE.
    MOVE WS-TYPE (1:20) TO LS-TYPE.
    EXIT METHOD.
END METHOD "ReadType".
EJECT
*****
*****
* Method ReadCost gets the cost of the item to be processed.*
*****
IDENTIFICATION DIVISION.
METHOD-ID. "ReadCost".

DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-EDIT-FLAG PIC X.
01 WS-COST-WORK PIC X(6).

```

```

*****
*   Define the linkage attributes.   *
*****

LINKAGE SECTION.
01  LS-COST                      PIC 999V99.

PROCEDURE DIVISION
    RETURNING  LS-COST.

    MOVE LOW-VALUE TO WS-EDIT-FLAG.
    PERFORM UNTIL WS-EDIT-FLAG = HIGH-VALUE
        DISPLAY "Enter the cost: "
        ACCEPT WS-COST-WORK          FROM SYSIN
        COMPUTE LS-COST = FUNCTION NUMVAL (WS-COST-WORK)
        IF LS-COST NUMERIC
            THEN MOVE HIGH-VALUE TO WS-EDIT-FLAG
        ELSE
            DISPLAY "Cost is not numeric - try again "
            DISPLAY "and get it right this time!!!"
        END-IF
    END-PERFORM.
    EXIT METHOD.
END METHOD "ReadCost".
EJECT

*****
*****
*   Method WriteMessage lets the system user know if the   *
*   requested action was successful.   *
*****

IDENTIFICATION DIVISION.
METHOD-ID. "WriteMessage".

DATA DIVISION.

*****
*   Define the linkage attributes.   *
*****

LINKAGE SECTION.
01  LS-FLAG                      PIC X.

PROCEDURE DIVISION
    USING      LS-FLAG.

    IF LS-FLAG = "0"
        THEN DISPLAY USER-ACTION "successful "
    ELSE
        DISPLAY USER-ACTION "failed "
    END-IF.
    EXIT METHOD.
END METHOD "WriteMessage".
EJECT

*****
*****
*   Method WriteOutput displays the order number and cost   *
*   to the system user.   *
*****

IDENTIFICATION DIVISION.
METHOD-ID. "WriteOutput".

DATA DIVISION.

```



```

END METHOD "WriteBottle".
EJECT
*****
*****
*   Method ReadProcess gets the processing request from the   *
*   system user.                                             *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "ReadProcess".

DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-EDIT-FLAG                      PIC X.

*****
*   Define the linkage attributes.                           *
*****
LINKAGE SECTION.
01 LS-PROCESS                      PIC X(10).

PROCEDURE DIVISION
RETURNING LS-PROCESS.

    MOVE LOW-VALUE TO WS-EDIT-FLAG.
    PERFORM UNTIL WS-EDIT-FLAG NOT = LOW-VALUE
        DISPLAY "Enter process desired: new, status, exit: "
        ACCEPT USER-ACTION                      FROM SYSIN
        MOVE FUNCTION UPPER-CASE (USER-ACTION) TO USER-ACTION
        MOVE USER-ACTION                      TO LS-PROCESS

    EVALUATE USER-ACTION (1:3)
        WHEN "NEW"
            MOVE HIGH-VALUE TO WS-EDIT-FLAG
        WHEN "STA"
            MOVE HIGH-VALUE TO WS-EDIT-FLAG
        WHEN "EXI"
            MOVE HIGH-VALUE TO WS-EDIT-FLAG
        WHEN OTHER
            DISPLAY "Requested process was " USER-ACTION
            DISPLAY "Wrong! Get it right this time!!!"
    END-EVALUATE
    END-PERFORM.
    EXIT METHOD.
END METHOD "ReadProcess".
EJECT
*****
*****
*   Method ReadOrder gets the order number from the system   *
*   user.                                                     *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "ReadOrder".

DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-EDIT-FLAG                      PIC X.
01 WS-ORDER                      PIC X(5).
01 WS-ORDER-9                      PIC 9(5).

*****

```



```

*      Define the linkage attributes.
*****
LINKAGE SECTION.
01  LS-ORDER                      PIC X(5).

PROCEDURE DIVISION                RETURNING  LS-ORDER.

    MOVE LOW-VALUE TO WS-EDIT-FLAG.
    PERFORM UNTIL WS-EDIT-FLAG = HIGH-VALUE
        DISPLAY "Enter the order number: "
        ACCEPT WS-ORDER                      FROM SYSIN
        COMPUTE WS-ORDER-9 = FUNCTION NUMVAL (WS-ORDER)
        MOVE WS-ORDER-9    TO    LS-ORDER
        IF LS-ORDER NUMERIC
            THEN MOVE HIGH-VALUE TO WS-EDIT-FLAG
        ELSE
            DISPLAY "Order number is not numeric - try again "
            DISPLAY "and get it right this time!!!"
        END-IF
    END-PERFORM.

    EXIT METHOD.
END METHOD "ReadOrder".
EJECT
*****
*****
*      Method WriteStatus displays the inventory attributes and
*      out-of-stock items that are in the order.
*****
IDENTIFICATION DIVISION.
METHOD-ID. "WriteStatus".

DATA DIVISION.

WORKING-STORAGE SECTION.

01  LS-INDEX                      PIC 9(4)    COMP.

*****
*      Define the linkage attributes.
*****
LINKAGE SECTION.
01  LS-OUT-ITEMS.
    05  LS-OUT-COUNT              PIC S9(4).
    05  LS-OUT-ITEM              OCCURS 1 to 64 TIMES
                                DEPENDING ON LS-OUT-COUNT.
        10  LSO-TYPE              PIC X(20).
        10  LSO-COST              PIC 999V99.

PROCEDURE DIVISION                USING      LS-OUT-ITEMS.

    IF LS-OUT-COUNT > 0
        THEN DISPLAY "LIST OUT OF STOCK ITEMS: "
                INVOKE SELF "WriteBottle"
                        USING LS-OUT-ITEMS
    ELSE
        DISPLAY "ALL ITEMS IN STOCK!".
    EXIT METHOD.
END METHOD "WriteStatus".

```

```

EJECT
*****
*****
*   Method WriteOutCount lets the system user know how many   *
*   old orders were checked.                                   *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "WriteOutCount".

DATA DIVISION.

*****
*   Define the linkage attributes.                               *
*****
LINKAGE SECTION.
01  LS-COUNT                                PIC S9(4)          COMP.

PROCEDURE DIVISION
    USING      LS-COUNT.

    DISPLAY LS-COUNT " orders checked".
    EXIT METHOD.
END METHOD "WriteOutCount".
EJECT
*****
*****
*   Method WriteLost writes an error message when the order   *
*   doesn't exist on the order file.                           *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "WriteLost".

DATA DIVISION.

*****
*   Define the linkage attributes.                               *
*****
LINKAGE SECTION.
01  LS-ORDER-NUMBER                        PIC 9(5).

PROCEDURE DIVISION
    USING      LS-ORDER-NUMBER.

    DISPLAY LS-ORDER-NUMBER " not on order file".
    EXIT METHOD.
END METHOD "WriteLost".
SKIP3
SKIP3
*****
*   End object definition and class UserInterface.             *
*****
END CLASS "UserInterface".

```

---

## G.4 Meta Old Order Class Code

```

process pgmname(longmixed) test
  IDENTIFICATION DIVISION.

  *****
  *   Class MetaOldOrder: Inherits from SOMClass           *
  *                               in the SOM Class Library.   *
  *****

  CLASS-ID.    MetaOldOrder    INHERITS SOMClass.

  *****
  *   Class MetaOldOrder contains the following methods:    *
  *   somDefaultInit -   Initializes a MetaOldOrder object. *
  *   CreateOldOrder -   Creates old order objects.         *
  *   CountOldOrders -   Counts old order objects.          *
  *****

  ENVIRONMENT DIVISION.

  *****
  *   Define which classes will be used by the methods in   *
  *   this class.                                           *
  *****

  CONFIGURATION SECTION.
  REPOSITORY.
      CLASS MetaOldOrder          IS "MetaOldOrder"
      CLASS OldOrder              IS "OldOrder"
      CLASS SOMClass              IS "SOMClass".

  DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 STATUS-COUNT                PIC S9(4)          COMP.
  EJECT

  *****
  *   Method somDefaultInit initializes the MetaOldOrder object.*
  *****

  IDENTIFICATION DIVISION.
  METHOD-ID. "somDefaultInit"    OVERRIDE.

  PROCEDURE DIVISION.

  *****
  *   Initialize the status counter.                         *
  *****
      MOVE ZERO TO STATUS-COUNT.
      EXIT METHOD.
  END METHOD "somDefaultInit".
  EJECT

  *****
  *   Method CreateOldOrder creates an oldorder object to be *
  *   counted.                                               *
  *****

  IDENTIFICATION DIVISION.
  METHOD-ID. "CreateOldOrder".

  DATA DIVISION.

```

```

*****
*   Define the linkage attributes.   *
*****

LINKAGE SECTION.
01 LS-ORDER-NUMBER          PIC 9(5).
01 LS-RETURN-PARMS.
    05 univObj              USAGE OBJECT REFERENCE.
    05 LS-CHECK-PARMS.
        10 LS-LOST-FLAG      PIC X.
        10 LS-OUT-ITEMS.
            15 LS-OUT-COUNT    PIC S9(4).
            15 LS-OUT-ITEM     OCCURS 1 TO 64
                                DEPENDING ON LS-OUT-COUNT
                                INDEXED BY LS-INDEX.
        20 LSO-TYPE          PIC X(20).
        20 LSO-COST          PIC 999V99.

PROCEDURE DIVISION
                                USING      LS-ORDER-NUMBER
                                RETURNING   LS-RETURN-PARMS.

    IF LS-ORDER-NUMBER > 0
        THEN INVOKE SELF "somNew"      RETURNING univObj
        INVOKE univObj "CheckItems"
                                USING LS-ORDER-NUMBER
                                RETURNING LS-CHECK-PARMS
        ADD 1 TO STATUS-COUNT
    END-IF.

EXIT METHOD.
END METHOD "CreateOldOrder".
EJECT

*****
*****
*   Method CountOldOrders returns the status-count field.   *
*****
*****

IDENTIFICATION DIVISION.
METHOD-ID. "CountOldOrders".

DATA DIVISION.

*****
*****
*   Define the linkage attributes.   *
*****

LINKAGE SECTION.
01 LS-STATUS-COUNT          PIC S9(4)      COMP.

PROCEDURE DIVISION
                                RETURNING   LS-STATUS-COUNT.

    MOVE STATUS-COUNT TO LS-STATUS-COUNT.

EXIT METHOD.
END METHOD "CountOldOrders".
SKIP3
SKIP3

*****
*   End object definition and class MetaOldOrder.   *
*****
END CLASS MetaOldOrder.

```

---

## G.5 FileRW Class Code

```
process pgmname(mixed) test
  IDENTIFICATION DIVISION.

  *****
  *   Class FileRW   : Inherits from SOMObject           *
  *                   in the SOM Class Library.           *
  *****

  CLASS-ID.  "FileRW"  INHERITS SOMObject.

  *****
  *   Class FileRW contains the following methods:       *
  *   XternOrder    -   Externalizes an order to a flat   *
  *                   file.                               *
  *   XReadOrder    -   Reads the order record from a flat *
  *                   file and returns it to the invoker.  *
  *****

  ENVIRONMENT DIVISION.

  *****
  *   Define which classes will be used by the methods in *
  *   this class.                                         *
  *****

  CONFIGURATION SECTION.
  REPOSITORY.
      CLASS SOMObject          IS "SOMObject"
      CLASS WineOrder          IS "WineOrder".

  *****
  *   Define the WineOrder Object.                       *
  *****

  DATA DIVISION.
  WORKING-STORAGE SECTION.

  PROCEDURE DIVISION.

  *****
  *   Method XternOrder writes the order to a flat file.  *
  *****

  IDENTIFICATION DIVISION.
  METHOD-ID.  "XternOrder".

  ENVIRONMENT DIVISION.
  INPUT-OUTPUT SECTION.
  FILE-CONTROL.
      SELECT ORDERS          ASSIGN TO   ORDERS
      FILE STATUS IS WS-STATUS-FLAG
      ORGANIZATION IS LINE SEQUENTIAL.

  DATA DIVISION.
  FILE SECTION.
  FD  ORDERS EXTERNAL
      RECORD CONTAINS 255.
```

```

01 ORDER-RECORD                PIC X(255).

WORKING-STORAGE SECTION.
01 WS-STATUS-FLAG              PIC XX.
01 WS-ORDER-RECORD.
    05 WS-ORDER-NUMBER         PIC X(5).
    05 WS-ORDER-DATE           PIC X(8).
    05 FILLER                  PIC XXX.
    05 WS-ITEMS.
        10 WS-ORDER-COUNT      PIC S9(4).
        10 WS-ORDER-ITEM      OCCURS 1 TO 64
                                DEPENDING ON WS-ORDER-COUNT
                                INDEXED BY WS-INDEX.
        15 WSO-TYPE            PIC X(20).
        15 WSO-COST            PIC 999V99.

LINKAGE SECTION.
01 orderObj                    USAGE OBJECT REFERENCE WineOrder.

PROCEDURE DIVISION    USING orderObj.

*****
*   Open the flat file for output.   *
*****
    OPEN OUTPUT ORDERS.
    MOVE SPACES TO WS-ORDER-RECORD.

*****
*   Get all the instance data for the order object.   *
*****
    INVOKE orderObj "GetInstanceData"
                RETURNING WS-ORDER-RECORD.

*****
*   Write the record.   *
*****
    WRITE  ORDER-RECORD      FROM WS-ORDER-RECORD.

*****
*   Close the order file after writing the record to it.   *
*****
    CLOSE ORDERS.

    EXIT METHOD.
    END METHOD "XternOrder".
    EJECT

*****
*   Method XReadOrder reads the order from a flat file.   *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "XReadOrder".

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT ORDERS      ASSIGN TO  ORDERS
    FILE STATUS IS WS-STATUS-FLAG
    ORGANIZATION IS LINE SEQUENTIAL.

```

```

DATA DIVISION.
FILE SECTION.
FD ORDERS EXTERNAL
   RECORD CONTAINS 255.
01 ORDER-RECORD          PIC X(255).

WORKING-STORAGE SECTION.
01 WS-STATUS-FLAG        PIC XX.
01 WS-EOF-FLAG           PIC X.

LINKAGE SECTION.
01 LS-ORDER              PIC X(5).
01 LS-ORDER-RECORD.
   05 LS-ORDER-NUMBER    PIC X(5).
   05 LS-ORDER-DATE      PIC X(8).
   05 FILLER             PIC XXX.
   05 LS-ORDER-COUNT     PIC S9(4).
   05 LS-ORDER-ITEM      OCCURS 1 TO 64
                        DEPENDING ON LS-ORDER-COUNT
                        INDEXED BY LS-INDEX.
   10 LSO-TYPE           PIC X(20).
   10 LSO-COST           PIC 999V99.

```

```

PROCEDURE DIVISION    USING LS-ORDER
                      RETURNING LS-ORDER-RECORD.

```

```

*****
*   Open the flat file for input; initialize eof flag.   *
*****
      OPEN INPUT  ORDERS.
      MOVE LOW-VALUE TO WS-EOF-FLAG.

*****
*   Read until the requested order is found on the file. *
*****
      PERFORM UNTIL WS-EOF-FLAG = HIGH-VALUE
        OR LS-ORDER-NUMBER = LS-ORDER
        READ ORDERS INTO LS-ORDER-RECORD
          AT END MOVE HIGH-VALUE TO WS-EOF-FLAG
        NOT AT END
          IF LS-ORDER-NUMBER = LS-ORDER
            THEN CONTINUE
          END-IF
        END-READ
      END-PERFORM.

*****
*   Close the order file after reading the record.      *
*****
      CLOSE ORDERS.

      EXIT METHOD.
      END METHOD "XReadOrder".
      SKIP3
      SKIP3
*****
*   End object definition and class FileRW.             *
*****

```

```
*****
END CLASS "FileRW".
```

## G.6 Bottle Class Code

```
process pgmname(mixed) test
  IDENTIFICATION DIVISION.

  *****
  *   Class WineBottle : Inherits from somf_MCollectible   *
  *   in the SOM Class Library.                             *
  *****

  CLASS-ID.  "WineBottle"  INHERITS somf-MCollectible.

  *****
  *   Class WineBottle contains the following methods:     *
  *   somfIsEqual    -   Provides SOM a method to see if two *
  *                   objects are equivalent.               *
  *   SetCost        -   Sets the cost of a WineBottle object *
  *                   based on a given object reference.     *
  *   SetType        -   Sets the type of a WineBottle object *
  *                   based on a given object reference.     *
  *   GetCost        -   Retrieves the cost of a WineBottle  *
  *                   object based on a given object         *
  *                   reference.                             *
  *   GetType        -   Retrieves the type of a WineBottle  *
  *                   object based on a given object         *
  *                   reference.                             *
  *   GetStatus      -   Retrieves the inventory status of   *
  *                   a wine bottle based on a given object *
  *                   reference.                             *
  *****

  ENVIRONMENT DIVISION.

  *****
  *   Define which classes will be used by the methods in   *
  *   this class.                                           *
  *****

  CONFIGURATION SECTION.
  REPOSITORY.
    CLASS WineBottle      IS "WineBottle"
    CLASS somf-MCollectible IS "somf_MCollectible".

  *****
  *   Define the WineBottle Object.                         *
  *****

  DATA DIVISION.
  WORKING-STORAGE SECTION.

  *****
  *   Define the instance data of the WineBottle Object.   *
  *****

  01 WINEBOTTLE-OBJECT.
    05 WINE-TYPE          PIC X(20).
```



```

05 WINE-COST          PIC 999V99.
EJECT

```

PROCEDURE DIVISION.

```

*****
*****
*   Method somfIsEqual provides SOM a method to see if two   *
*   bottle objects are equivalent. In our case, if their     *
*   types and costs are the same, we consider them equal.    *
*****

```

IDENTIFICATION DIVISION.

METHOD-ID. "somfIsEqual" OVERRIDE.

DATA DIVISION.

LOCAL-STORAGE SECTION.

```

01 ITEMTYPE          PIC X(20).
01 ITEMCOST          PIC 999V99.

```

LINKAGE SECTION.

```

01 LS-EV             USAGE POINTER.
01 theBottle         Usage Object Reference WineBottle.
01 theEqualFlag      PIC X.

```

```

PROCEDURE DIVISION          USING BY VALUE LS-EV
                                BY VALUE theBottle
                                RETURNING          theEqualFlag.

```

```

*****
*   Get the type and cost of the bottle object               *
*****
      INVOKE theBottle  "GetType"  RETURNING ITEMTYPE.
      INVOKE theBottle  "GetCost"  RETURNING ITEMCOST.

```

```

*****
*   Get those just obtained to the attributes of this       *
*   instance. If they are equal, set the equality flag       *
*   to a binary 1, else set it to a low-value.              *
*****

```

```

      IF (WINE-TYPE = ITEMTYPE) AND
         (WINE-COST = ITEMCOST)
         THEN MOVE HIGH-VALUE TO theEqualFlag
      ELSE
         MOVE LOW-VALUE TO theEqualFlag.

```

EXIT METHOD.

END METHOD "somfIsEqual".

EJECT

```

*****
*****
*   Method GetType Gets the type of a WineBottle based on the *
*   object reference of the WineBottle.                      *
*****

```

IDENTIFICATION DIVISION.

METHOD-ID. "GetType".

DATA DIVISION.

WORKING-STORAGE SECTION.

```

*****
*   Define the linkage attributes.                               *
*****

LINKAGE SECTION.
01  LS-TYPE                                PIC X(20).

PROCEDURE DIVISION                                RETURNING  LS-TYPE.

*****
*   Move data to the LINKAGE SECTION.                           *
*****

MOVE WINE-TYPE TO LS-TYPE.

EXIT METHOD.
END METHOD "GetType".
EJECT

*****
*   Method GetCost Gets the COST of a WineBottle based on the *
*   object reference of the WineBottle.                         *
*****

IDENTIFICATION DIVISION.
METHOD-ID. "GetCost".

DATA DIVISION.
WORKING-STORAGE SECTION.

*****
*   Define the linkage attributes.                               *
*****

LINKAGE SECTION.
01  LS-COST                                PIC 999V99.

PROCEDURE DIVISION                                RETURNING  LS-COST.

*****
*   Move data to the LINKAGE SECTION.                           *
*****

MOVE WINE-COST TO LS-COST.

EXIT METHOD.
END METHOD "GetCost".
EJECT

*****
*   Method SetType Sets the type of a WineBottle based on the *
*   object reference of the WineBottle.                         *
*****

IDENTIFICATION DIVISION.
METHOD-ID. "SetType".

DATA DIVISION.
WORKING-STORAGE SECTION.

*****
*   Define the linkage attributes.                               *
*****

LINKAGE SECTION.
01  LS-TYPE                                PIC X(20).

```

```

PROCEDURE DIVISION                                USING      LS-TYPE.

*****
*      Move data to the LINKAGE SECTION.          *
*****
      MOVE LS-TYPE TO WINE-TYPE.

      EXIT METHOD.
      END METHOD "SetType".
      EJECT
*****
*****
*      Method SetCost Sets the COST of a WineBottle based on the *
*      object reference of the WineBottle.          *
*****
      IDENTIFICATION DIVISION.
      METHOD-ID. "SetCost".

      DATA DIVISION.
      WORKING-STORAGE SECTION.

*****
*      Define the linkage attributes.              *
*****
      LINKAGE SECTION.
      01 LS-COST                                PIC 999V99.

      PROCEDURE DIVISION                                USING      LS-COST.

*****
*      Move data to the LINKAGE SECTION.          *
*****
      MOVE LS-COST TO WINE-COST.

      EXIT METHOD.
      END METHOD "SetCost".
      EJECT
*****
*****
*      Method GetStatus gets the inventory status of an item    *
*      that was ordered.          *
*****
      IDENTIFICATION DIVISION.
      METHOD-ID. "GetStatus".

      DATA DIVISION.
      WORKING-STORAGE SECTION.
      01 WS-STATUS-WORK                        PIC 9(5).
      01 WS-STATUS-MOD                        PIC 9.
      01 WS-RANDOM-WORK                      PIC 9V9(5).

*****
*      Define the linkage attributes.              *
*****
      LINKAGE SECTION.
      01 LS-STATUS                                PIC X.

      PROCEDURE DIVISION                                RETURNING    LS-STATUS.

```

```

*****
*   We aren't reading the inventory quantity from an      *
*   external file, so we need to generate our order status *
*   in here. For starters, we'll assume that we have a 50% *
*   chance of the item being in stock.                     *
*****
      COMPUTE WS-RANDOM-WORK = FUNCTION RANDOM.
      COMPUTE WS-STATUS-WORK = WS-RANDOM-WORK * 10000.
      DIVIDE WS-STATUS-WORK BY 2 GIVING WS-STATUS-WORK
          REMAINDER WS-STATUS-MOD.
*****
*   If the generated number is even, set the status flag to *
*   0 which means out-of-stock; else set it to 1, or in-stock.*
*****
      IF WS-STATUS-MOD = 0
          THEN MOVE "0" TO LS-STATUS
      ELSE
          MOVE "1" TO LS-STATUS.

*****
*   Move data to the LINKAGE SECTION.                      *
*****

      EXIT METHOD.
      END METHOD "GetStatus".
      SKIP3
      SKIP3
*****
*   End object definition and class WineBottle.           *
*****
      END CLASS "WineBottle".

```

---

## G.7 Order Class Code

```

process pgmname(mixed) test
    IDENTIFICATION DIVISION.

*****
*   Class WineOrder : Inherits from SOMObject             *
*   in the SOM Class Library.                             *
*****

      CLASS-ID.  "WineOrder" INHERITS SOMObject.

*****
*   Class WineOrder contains the following methods:      *
*   somDefaultInit -   Initializes a WineOrder object.   *
*   somFree          -   Frees bottles, collection, and order. *
*   SetOrderNumber   -   Sets the number of a WineOrder object *
*                       based on a given object reference.   *
*   SetOrderDate     -   Sets the date of a WineOrder object *
*                       based on a given object reference.   *
*   AddBottle        -   Adds a bottle object to the order  *
*   RemoveBottle     -   Removes a bottle object from the   *
*                       order.                                *
*   CalculateCost    -   Computes the cost of the bottle    *
*                       objects in the order.               *

```

```

*   DescribeOrder - Lists the contents of the bottles      *
*   collected in the order.                                *
*   GetOrderNumber - Retrieves the number of a WineOrder  *
*   object.                                                *
*   GetOrderDate   - Retrieves the date of a WineOrder    *
*   object.                                                *
*   SetInstanceData- Sets all the attributes of an order  *
*   object.                                                *
*   GetInstanceData- Gets all the attributes of an order  *
*   object.                                                *
*   GetEV          - Retrieves the SOM environment        *
*   variable.                                              *
*   GetList        - Retrieves the SOM list for the        *
*   collected order items.                                *
*   GetIterator    - Retrieves the SOM iterator for the    *
*   collected order items.                                *
*****

```

#### ENVIRONMENT DIVISION.

```

*****
*   Define which classes will be used by the methods in    *
*   this class.                                            *
*****

```

#### CONFIGURATION SECTION.

##### REPOSITORY.

```

CLASS SOMObject      IS "SOMObject"
CLASS SOMCollection  IS "somf_TSet"
CLASS SOMIterator    IS "somf_TSetIterator"
CLASS WineBottle     IS "WineBottle".

```

```

*****
*   Define the WineOrder Object.                          *
*****

```

#### DATA DIVISION.

##### WORKING-STORAGE SECTION.

```

01 WS-EV                      USAGE POINTER.

```

```

*****
*   Define the instance data of the WineOrder Object.      *
*****

```

```

01 WINEORDER-OBJECT.
   05 WINEORDER-NUMBER      PIC X(5).
   05 WINEORDER-DATE        PIC X(8).
   05 WINEORDER-LIST  USAGE OBJECT REFERENCE SOMCollection.

```

```

*****
*   Define an iterator for use on the wineorder-list.      *
*****

```

```

01 WINEORDER-ITERATOR  USAGE OBJECT REFERENCE SOMIterator.
   EJECT

```

#### PROCEDURE DIVISION.

```

*****
*   The overridden method somDefaultInit initializes the   *

```

```

*      WineOrder instance, and creates the collection to be      *
*      used in the order.                                         *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "somDefaultInit"      OVERRIDE.

DATA DIVISION.

PROCEDURE DIVISION.

*****
*      Initialize the SOM global environment variable.           *
*****
      CALL "somGetGlobalEnvironment" RETURNING WS-EV.
*****
*      Now initialize an empty collection for us to add bottles  *
*      into with the addBottle method.                           *
*****
      INVOKE SOMCollection "somNew"
                      RETURNING WINEORDER-LIST.

*****
*      Instantiate an iterator object and associate it with the  *
*      collection.                                               *
*****
      INVOKE WINEORDER-LIST "somfCreateIterator"
                      USING      BY VALUE WS-EV
                      RETURNING WINEORDER-ITERATOR.

*****
*      EXIT and END the method.                                   *
*****
      EXIT METHOD.
      END METHOD "somDefaultInit".
      EJECT
*****
*      The overridden method somFree      destroys the bottle   *
*      objects created, the collection object, and the order    *
*      object.                                                   *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "somFree"      OVERRIDE.

DATA DIVISION.
WORKING-STORAGE SECTION.
01 CollectedBottle      USAGE OBJECT REFERENCE WineBottle.
01 ITEM-COUNT            PIC S9(8)      COMP.

PROCEDURE DIVISION.

*****
*      Get the collected objects.                                 *
*****
      INVOKE WINEORDER-LIST "somfDeleteAll"
                      USING      BY VALUE WS-EV.

*****

```

```

*      Free the list and iterator objects                                     *
*****
      INVOKE WINEORDER-ITERATOR "somFree".

      INVOKE WINEORDER-LIST "somFree".

*****
*      Free thyself...Use SUPER so we don't recurse back into          *
*      this method.                                                       *
*****
      INVOKE SUPER "somFree".

*****
*      EXIT and END the method.                                           *
*****
      EXIT METHOD.
      END METHOD "somFree".
      EJECT

*****
*****
*      Method GetOrderNumber gets the number of WineOrder based      *
*      on the object reference of the WineOrder.                       *
*****

      IDENTIFICATION DIVISION.
      METHOD-ID. "GetOrderNumber".

      DATA DIVISION.
      WORKING-STORAGE SECTION.

*****
*****
*      Define the linkage attributes.                                     *
*****

      LINKAGE SECTION.
      01  LS-ORDERNUMBER                                PIC X(5).

      PROCEDURE DIVISION                                RETURNING  LS-ORDERNUMBER.

*****
*      Move data to the LINKAGE SECTION.                                 *
*****
      MOVE WINEORDER-NUMBER TO LS-ORDERNUMBER.

      EXIT METHOD.
      END METHOD "GetOrderNumber".
      EJECT

*****
*****
*      Method GetOrderDate gets the date of a WineOrder based        *
*      on the object reference of the WineOrder.                       *
*****

      IDENTIFICATION DIVISION.
      METHOD-ID. "GetOrderDate".

      DATA DIVISION.
      WORKING-STORAGE SECTION.

*****
*****
*      Define the linkage attributes.                                     *
*****

```

```

LINKAGE SECTION.
01 LS-ORDERDATE                                PIC X(8).

PROCEDURE DIVISION                                RETURNING    LS-ORDERDATE.

*****
*   Move data to the LINKAGE SECTION.             *
*****
        MOVE WINEORDER-DATE TO LS-ORDERDATE.

        EXIT METHOD.
        END METHOD "GetOrderDate".
        EJECT
*****
*****
*   Method SetOrderNumber sets the number of WineOrder based *
*   on the object reference of the WineOrder.             *
*****
        IDENTIFICATION DIVISION.
        METHOD-ID. "SetOrderNumber".

        DATA DIVISION.
        WORKING-STORAGE SECTION.

*****
*   Define the linkage attributes.                 *
*****
        LINKAGE SECTION.
        01 LS-ORDERNUMBER                        PIC X(5).

        PROCEDURE DIVISION                                USING      LS-ORDERNUMBER.

*****
*   Move data from the LINKAGE SECTION.           *
*****
        MOVE LS-ORDERNUMBER TO WINEORDER-NUMBER.

        EXIT METHOD.
        END METHOD "SetOrderNumber".
        EJECT
*****
*****
*   Method SetOrderDate sets the date of a WineOrder based *
*   on the object reference of the WineOrder.             *
*****
        IDENTIFICATION DIVISION.
        METHOD-ID. "SetOrderDate".

        DATA DIVISION.
        WORKING-STORAGE SECTION.

*****
*   Define the linkage attributes.                 *
*****
        LINKAGE SECTION.
        01 LS-ORDERDATE                                PIC X(8).

        PROCEDURE DIVISION                                USING      LS-ORDERDATE.

```



```

*****
*   Move data from the LINKAGE SECTION.   *
*****
        MOVE LS-ORDERDATE TO WINEORDER-DATE.

        EXIT METHOD.
    END METHOD "SetOrderDate".
    EJECT
*****
*****
*   Method DescribeOrder describes the order contents.   *
*****
    IDENTIFICATION DIVISION.
    METHOD-ID. "DescribeOrder".

    DATA DIVISION.

    LOCAL-STORAGE SECTION.
    01 CollectedBottle      USAGE OBJECT REFERENCE WineBottle.
    01 WS-TYPE                PIC X(20).
    01 WS-COST                PIC 999V99.
    01 ITEM-COUNT             PIC S9(8)   COMP.

    LINKAGE SECTION.
    01 LS-ITEMS.
        05 LS-ITEM-COUNT      PIC S9(4).
        05 LS-ITEM            OCCURS 1 TO 64 TIMES
                               DEPENDING ON LS-ITEM-COUNT
                               INDEXED BY   LS-INDEX.
        10 LS-TYPE            PIC X(20).
        10 LS-COST            PIC 999V99.

    PROCEDURE DIVISION      RETURNING LS-ITEMS.

    *****
    *   Get the count of the number of items in the collection.   *
    *****
        INVOKE WINEORDER-LIST "somfCount"
                               USING      BY VALUE WS-EV
                               RETURNING   ITEM-COUNT.
        MOVE ITEM-COUNT TO LS-ITEM-COUNT.

    *****
    *   Get the first one in the collection.   *
    *****
        IF ITEM-COUNT > 0
            THEN SET LS-INDEX TO 1
                INVOKE WINEORDER-ITERATOR "somfFirst"
                                         USING      BY VALUE WS-EV
                                         RETURNING CollectedBottle
                PERFORM GET-TYPE-N-COST
            END-IF.

    *****
    *   Get the rest...   *
    *****
        SUBTRACT 1 FROM ITEM-COUNT.
        IF ITEM-COUNT > 0
            THEN PERFORM ITEM-COUNT TIMES

```

```

        SET LS-INDEX UP BY 1
        INVOKE WINEORDER-ITERATOR "somfNext"
                USING      BY VALUE WS-EV
                RETURNING CollectedBottle
        PERFORM GET-TYPE-N-COST
    END-PERFORM
END-IF.

*****
*   Exit and end the method.                               *
*****
EXIT METHOD.

*****
*   Invoke the gettype and getcost methods on the bottle   *
*   object and move the returned attributes to the table.   *
*****
GET-TYPE-N-COST.
    INVOKE CollectedBottle "GetType" RETURNING WS-TYPE.
    MOVE WS-TYPE TO LS-TYPE (LS-INDEX).
    INVOKE CollectedBottle "GetCost" RETURNING WS-COST.
    MOVE WS-COST TO LS-COST (LS-INDEX).

END METHOD "DescribeOrder".
EJECT
*****
*****
*   Method CalculateCost computes the cost of the order.   *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "CalculateCost".

DATA DIVISION.
WORKING-STORAGE SECTION.
01 CollectedBottle    USAGE OBJECT REFERENCE WineBottle.
01 ITEM-COUNT          PIC S9(8)    COMP.
01 WS-COST             PIC 999V99.

*****
*   Define the linkage attributes.                           *
*****
LINKAGE SECTION.
01 LS-COST             PIC 9(7)V99.

PROCEDURE DIVISION
    RETURNING LS-COST.

*****
*   Initialize the accumulator for the total cost.         *
*****
MOVE ZERO TO LS-COST.

*****
*   Get the count of the number of items in the collection. *
*****
    INVOKE WINEORDER-LIST "somfCount"
        USING      BY VALUE WS-EV
        RETURNING ITEM-COUNT.

*****

```

```

*      Get the first one in the collection.                                     *
*****
      IF ITEM-COUNT > 0
          INVOKE WINEORDER-ITERATOR "somfFirst"
              USING      BY VALUE WS-EV
              RETURNING CollectedBottle
          PERFORM GET-COST
      END-IF.

*****
*      Get the rest...                                                         *
*****
      SUBTRACT 1 FROM ITEM-COUNT.
      IF ITEM-COUNT > 0
          THEN PERFORM ITEM-COUNT TIMES
              INVOKE WINEORDER-ITERATOR "somfNext"
                  USING      BY VALUE WS-EV
                  RETURNING CollectedBottle
              PERFORM GET-COST
          END-PERFORM
      END-IF.

*****
*      EXIT the method and return.                                           *
*****
      EXIT METHOD.

*****
*      Invoke the getcost method on the bottle object and                   *
*      accumulate the cost.                                                  *
*****
      GET-COST.
      INVOKE CollectedBottle "GetCost" RETURNING WS-COST.
      ADD WS-COST TO LS-COST.

      END METHOD "CalculateCost".
      EJECT
*****
*      Method AddBottle adds a bottle of wine to the bottle                 *
*      collection in the wine order.                                         *
*****
      IDENTIFICATION DIVISION.
      METHOD-ID. "AddBottle".

      DATA DIVISION.
      WORKING-STORAGE SECTION.
      01 WS-BEFORE-COUNT                PIC S9(8)    COMP.
      01 WS-AFTER-COUNT                 PIC S9(8)    COMP.
      01 CollectedBottle                USAGE OBJECT REFERENCE WineBottle.

      01 theEqualFlag                   PIC X.
      01 ITEM-FOUND-FLAG                 PIC X.
      01 ITEM-COUNT                     PIC S9(8)    COMP.
      01 LOOP-COUNT                     PIC S9(8)    COMP.

*****
*      Define the linkage attributes.                                         *
*****

```

```

LINKAGE SECTION.
01 LS-BOTTLE          USAGE OBJECT REFERENCE WineBottle.
01 LS-PARMS.
    05 LS-ITEM-COUNT    PIC S9(8)    COMP.
    05 LS-FLAG          PIC X.

PROCEDURE DIVISION
                                USING      LS-BOTTLE
                                RETURNING  LS-PARMS.

    MOVE LOW-VALUE            TO ITEM-FOUND-FLAG.

*****
*   Get the count of items before adding the object.   *
*****
    INVOKE WINEORDER-LIST "somfCount"
                                USING      BY VALUE WS-EV
                                RETURNING  WS-BEFORE-COUNT.
    MOVE    WS-BEFORE-COUNT TO ITEM-COUNT.

*****
*   Get the first one in the collection.               *
*****
    IF ITEM-COUNT NOT = 0
        THEN INVOKE WINEORDER-ITERATOR "somfFirst"
                                USING      BY VALUE WS-EV
                                RETURNING  CollectedBottle
        PERFORM CHECK-EQUAL
    END-IF.

*****
*   Get the rest...                                   *
*****
    SUBTRACT 1 FROM ITEM-COUNT.
    IF ITEM-COUNT > 0
        THEN PERFORM VARYING LOOP-COUNT
            FROM 1 BY 1
            UNTIL LOOP-COUNT > ITEM-COUNT
                OR ITEM-FOUND-FLAG = HIGH-VALUE
            INVOKE WINEORDER-ITERATOR "somfNext"
                                USING      BY VALUE WS-EV
                                RETURNING  CollectedBottle
            PERFORM CHECK-EQUAL
        END-PERFORM
    END-IF.

*****
*   Add the bottle to the collection if it hasn't been *
*   added before.                                     *
*****
    IF ITEM-FOUND-FLAG = LOW-VALUE
        THEN INVOKE WINEORDER-LIST "somfAdd"
                                USING BY VALUE WS-EV
                                BY VALUE LS-BOTTLE.

*****
*   Get the count of items after adding the object.   *
*****
    INVOKE WINEORDER-LIST "somfCount"
                                USING      BY VALUE WS-EV

```

```

                                RETURNING WS-AFTER-COUNT.
MOVE WS-AFTER-COUNT TO LS-ITEM-COUNT.

*****
*   If the counts are the same the add failed.   *
*****
    IF WS-BEFORE-COUNT = WS-AFTER-COUNT
        THEN MOVE "1" TO LS-FLAG
    ELSE
        MOVE "0" TO LS-FLAG
    END-IF.

*****
*   EXIT the method and return.   *
*****
    EXIT METHOD.

*****
*   Invoke the somIsEqual method in the bottle object to   *
*   see if the objects are equal. Set a flag if they are.   *
*****
CHECK-EQUAL.
    INVOKE CollectedBottle "somIsEqual"
                                USING BY VALUE WS-EV
                                BY VALUE LS-BOTTLE
                                RETURNING theEqualFlag.
    IF theEqualFlag = HIGH-VALUE
        THEN MOVE HIGH-VALUE TO ITEM-FOUND-FLAG.

END METHOD "AddBottle".
EJECT

*****
*   Method RemoveBottle removes a bottle from the bottle   *
*   collection in the wine order.   *
*****
IDENTIFICATION DIVISION.
METHOD-ID. "RemoveBottle".

DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-BEFORE-COUNT PIC S9(8) COMP.
01 WS-AFTER-COUNT PIC S9(8) COMP.
01 CollectedBottle USAGE OBJECT REFERENCE WineBottle.
01 theEqualFlag PIC X.
01 ITEM-COUNT PIC S9(8) COMP.
01 LOOP-COUNT PIC S9(8) COMP.

*****
*   Define the linkage attributes.   *
*****
LINKAGE SECTION.
01 LS-BOTTLE USAGE OBJECT REFERENCE WineBottle.
01 LS-PARMS.
    05 LS-ITEM-COUNT PIC S9(8) COMP.
    05 LS-FLAG PIC X.

PROCEDURE DIVISION
    USING LS-BOTTLE
    RETURNING LS-PARMS.

```

```

*****
*   Get the count of items before the delete.   *
*****
        INVOKE WINEORDER-LIST "somfCount"
                USING      BY VALUE WS-EV
                RETURNING WS-BEFORE-COUNT.
        MOVE WS-BEFORE-COUNT TO ITEM-COUNT.

*****
*   Get the first one in the collection.   *
*****
        IF ITEM-COUNT NOT = 0
            THEN INVOKE WINEORDER-ITERATOR "somfFirst"
                    USING      BY VALUE WS-EV
                    RETURNING CollectedBottle
                PERFORM CHECK-EQUAL-N-REMOVE
        END-IF.

*****
*   Get the rest...   *
*****
        SUBTRACT 1 FROM ITEM-COUNT.
        IF ITEM-COUNT > 0
            THEN PERFORM VARYING LOOP-COUNT
                    FROM 1 BY 1
                    UNTIL LOOP-COUNT > ITEM-COUNT
                        OR theEqualFlag = HIGH-VALUE
                INVOKE WINEORDER-ITERATOR "somfNext"
                        USING      BY VALUE WS-EV
                        RETURNING CollectedBottle
                PERFORM CHECK-EQUAL-N-REMOVE
            END-PERFORM
        END-IF.

*****
*   Get the count of items after the delete.   *
*****
        INVOKE WINEORDER-LIST "somfCount"
                USING      BY VALUE WS-EV
                RETURNING WS-AFTER-COUNT.
        MOVE WS-AFTER-COUNT TO LS-ITEM-COUNT.

*****
*   If the counts are the same the delete failed.   *
*****
        IF WS-BEFORE-COUNT = WS-AFTER-COUNT
            THEN MOVE "1" TO LS-FLAG
        ELSE
            MOVE "0" TO LS-FLAG
        END-IF.

*****
*   EXIT the method and return.   *
*****
        EXIT METHOD.

CHECK-EQUAL-N-REMOVE.

```

```

        INVOKE CollectedBottle "somfIsEqual"
                USING BY VALUE WS-EV
                BY VALUE LS-BOTTLE
                RETURNING theEqualFlag.
*****
*   If we find one, remove it from the list.   *
*****
        IF theEqualFlag = HIGH-VALUE
            THEN INVOKE WINEORDER-LIST "somfRemove"
                USING BY VALUE WS-EV
                BY VALUE CollectedBottle
                INVOKE CollectedBottle "somFree".

    END METHOD "RemoveBottle".
    EJECT
*****
*   Method GetEV gets the SOM environment pointer.   *
*****
    IDENTIFICATION DIVISION.
    METHOD-ID. "GetEV".

    DATA DIVISION.
    WORKING-STORAGE SECTION.

*****
*   Define the linkage attributes.   *
*****
    LINKAGE SECTION.
    01 LS-EV                USAGE POINTER.

    PROCEDURE DIVISION
                RETURNING LS-EV.

*****
*   Move data to the LINKAGE SECTION.   *
*****
        SET LS-EV TO WS-EV.

    EXIT METHOD.
    END METHOD "GetEV".
    EJECT
*****
*   Method GetList gets the wineorder list collection.   *
*****
    IDENTIFICATION DIVISION.
    METHOD-ID. "GetList".

    DATA DIVISION.
    WORKING-STORAGE SECTION.

*****
*   Define the linkage attributes.   *
*****
    LINKAGE SECTION.
    01 LS-LIST                USAGE OBJECT REFERENCE SOMCollection.

    PROCEDURE DIVISION
                RETURNING LS-LIST.

```

```

*****
*   Move data to the LINKAGE SECTION.   *
*****
      SET LS-LIST TO WINEORDER-LIST.

      EXIT METHOD.
      END METHOD "GetList".
      EJECT
*****
*****
*   Method GetIterator gets the wineorder list iterator.   *
*****
      IDENTIFICATION DIVISION.
      METHOD-ID. "GetIterator".

      DATA DIVISION.
      WORKING-STORAGE SECTION.

*****
*   Define the linkage attributes.   *
*****
      LINKAGE SECTION.
      01 LS-ITERATOR USAGE OBJECT REFERENCE SOMIterator.

      PROCEDURE DIVISION          RETURNING    LS-ITERATOR.

*****
*   Move data to the LINKAGE SECTION.   *
*****
      SET LS-ITERATOR TO WINEORDER-ITERATOR.

      EXIT METHOD.
      END METHOD "GetIterator".
      EJECT
*****
*****
*   Method SetInstanceData sets all the attributes of an   *
*   order object from data passed in the linkage section.   *
*****
      IDENTIFICATION DIVISION.
      METHOD-ID. "SetInstanceData".

      DATA DIVISION.
      WORKING-STORAGE SECTION.
      01 WS-PARMS.
          05 ITEM-COUNT          PIC S9(8)          COMP.
          05 WS-FLAG             PIC X.
              88 SUCCESSFUL          VALUE "0".
              88 FAILURE            VALUE "1".
      01 bottleObj              USAGE OBJECT REFERENCE WineBottle.

*****
*   Define the linkage attributes.   *
*****
      LINKAGE SECTION.
      01 LS-ORDER.
          05 LS-ORDER-NUMBER      PIC X(5).
          05 LS-ORDER-DATE        PIC X(8).
          05 FILLER                PIC XXX.

```



```

05 LS-ORDER-COUNT          PIC S9(4).
05 LS-ORDER-ITEM           OCCURS 1 TO 64 TIMES
                           DEPENDING ON LS-ORDER-COUNT
                           INDEXED BY LS-INDEX.
    10 LSO-TYPE             PIC X(20).
    10 LSO-COST             PIC 999V99.

PROCEDURE DIVISION
    USING LS-ORDER.

*****
*   Move in the easy stuff...
*****
    INVOKE self "SetOrderNumber" USING LS-ORDER-NUMBER.
    INVOKE self "SetOrderDate"   USING LS-ORDER-DATE.

*****
*   And now the tricky stuff...
*****
    PERFORM VARYING LS-INDEX FROM 1 BY 1
        UNTIL LS-INDEX > LS-ORDER-COUNT
        INVOKE WineBottle "somNew" RETURNING bottleObj
        INVOKE bottleObj "SetType" USING LSO-TYPE (LS-INDEX)
        INVOKE bottleObj "SetCost" USING LSO-COST (LS-INDEX)
        INVOKE self "AddBottle"   USING bottleObj
        RETURNING WS-PARMS

    END-PERFORM.

EXIT METHOD.
END METHOD "SetInstanceData".
EJECT

*****
*   Method GetInstanceData retrieves all the attributes of an
*   order object and places them in the linkage section.
*****
IDENTIFICATION DIVISION.
METHOD-ID. "GetInstanceData".

DATA DIVISION.
WORKING-STORAGE SECTION.

*****
*   Define the linkage attributes.
*****
LINKAGE SECTION.
01 LS-ORDER.
    05 LS-ORDER-NUMBER          PIC X(5).
    05 LS-ORDER-DATE           PIC X(8).
    05 FILLER                  PIC XXX.
    05 LS-ITEMS.
        10 LS-ORDER-COUNT      PIC S9(4).
        10 LS-ORDER-ITEM      OCCURS 1 TO 64 TIMES
                                DEPENDING ON LS-ORDER-COUNT
                                INDEXED BY LS-INDEX.
            15 LSO-TYPE        PIC X(20).
            15 LSO-COST        PIC 999V99.

```

```

PROCEDURE DIVISION                                RETURNING  LS-ORDER.

    INVOKE self "GetOrderNumber" RETURNING LS-ORDER-NUMBER.
    INVOKE self "GetOrderDate"   RETURNING LS-ORDER-DATE.
    INVOKE self "DescribeOrder"  RETURNING LS-ITEMS.

    EXIT METHOD.
END METHOD "GetInstanceData".
    SKIP3
    SKIP3
*****
*      End object definition and class WineOrder.      *
*****
END CLASS "WineOrder".

```

---

## Glossary

The terms in this glossary are defined in accordance with their meaning in COBOL. These terms may or may not have the same meaning in other languages.

IBM is grateful to the American National Standards Institute (ANSI) for permission to reprint its definitions from the following publications:

- *American National Standard Programming Language COBOL, ANSI X3.23-1985* (Copyright 1985 American National Standards Institute, Inc.), which was prepared by Technical Committee X3J4, which had the task of revising American National Standard COBOL, X3.23-1974.
- *American National Dictionary for Information Processing Systems* (Copyright 1982 by the Computer and Business Equipment Manufacturers Association).

American National Standard definitions are preceded by an asterisk (\*).

### A

\* **abbreviated combined relation condition.** The combined condition that results from the explicit omission of a common subject or a common subject and common relational operator in a consecutive sequence of relation conditions.

**abend.** Abnormal termination of program.

\* **access mode.** The manner in which records are to be operated upon within a file.

\* **actual decimal point.** The physical representation, using the decimal point characters period (.) or comma (,), of the decimal point position in a data item.

\* **alphabet-name.** A user-defined word, in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION, that assigns a name to a specific character set and/or collating sequence.

\* **alphabetic character.** A letter or a space character.

\* **alphanumeric character.** Any character in the computer's character set.

**alphanumeric-edited character.** A character within an alphanumeric character-string that contains at least one B, 0 (zero), or / (slash).

\* **alphanumeric function.** A function whose value is composed of a string of one or more characters from the computer's character set.

\* **alternate record key.** A key, other than the prime record key, whose contents identify a record within an indexed file.

**ANSI (American National Standards Institute).** An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States.

\* **argument.** An identifier, a literal, an arithmetic expression, or a function-identifier that specifies a value to be used in the evaluation of a function.

\* **arithmetic expression.** An identifier of a numeric elementary item, a numeric literal, such identifiers and literals separated by arithmetic operators, two arithmetic expressions separated by an arithmetic operator, or an arithmetic expression enclosed in parentheses.

\* **arithmetic operation.** The process caused by the execution of an arithmetic statement, or the evaluation of an arithmetic expression, that results in a mathematically correct solution to the arguments presented.

\* **arithmetic operator.** A single character, or a fixed two-character combination that belongs to the following set:

Character	Meaning
+	addition
—	subtraction
*	multiplication
/	division
**	exponentiation

\* **arithmetic statement.** A statement that causes an arithmetic operation to be executed. The arithmetic statements are the ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT statements.

**array.** In &cel., an aggregate consisting of data objects, each of which may be uniquely referenced by subscripting. Roughly analogous to a COBOL table.

\* **ascending key.** A key upon the values of which data is ordered, starting with the lowest value of the key up to the highest value of the key, in accordance with the rules for comparing data items.

**ASCII.** American National Standard Code for Information Interchange. The standard code, using a coded character set consisting of 7-bit coded characters (8 bits including parity check), used for information interchange between data processing systems, data communication systems, and

associated equipment. The ASCII set consists of control characters and graphic characters.

**Note:** IBM has defined an extension to ASCII code (characters 128-255).

**assignment-name.** A name that identifies the organization of a COBOL file and the name by which it is known to the system.

\* **assumed decimal point.** A decimal point position that does not involve the existence of an actual character in a data item. The assumed decimal point has logical meaning with no physical representation.

\* **AT END condition.** A condition caused:

1. During the execution of a READ statement for a sequentially accessed file, when no next logical record exists in the file, or when the number of significant digits in the relative record number is larger than the size of the relative key data item, or when an optional input file is not present.
2. During the execution of a RETURN statement, when no next logical record exists for the associated sort or merge file.
3. During the execution of a SEARCH statement, when the search operation terminates without satisfying the condition specified in any of the associated WHEN phrases.

## B

**big-endian.** Default format used by the mainframe and the AIX workstation to store binary data. In this format, the least significant digit is on the highest address. Compare with "little-endian."

**binary item.** A numeric data item represented in binary notation (on the base 2 numbering system). Binary items have a decimal equivalent consisting of the decimal digits 0 through 9, plus an operational sign. The leftmost bit of the item is the operational sign.

**binary search.** A dichotomizing search in which, at each step of the search, the set of data elements is divided by two; some appropriate action is taken in the case of an odd number.

\* **block.** A physical unit of data that is normally composed of one or more logical records. For mass storage files, a block may contain a portion of a logical record. The size of a block has no direct relationship to the size of the file within which the block is contained or to the size of the logical record(s) that are either contained within the block or that overlap the block. The term is synonymous with physical record.

**breakpoint.** A place in a computer program, usually specified by an instruction, where its execution may

be interrupted by external intervention or by a monitor program.

**Btrieve.** A key-indexed record management system that allows applications to manage records by key value, sequential access method, or random access method. IBM COBOL supports COBOL sequential and indexed file I-O language through Btrieve.

**buffer.** A portion of storage used to hold input or output data temporarily.

**built-in function.** See "intrinsic function".

**byte.** A string consisting of a certain number of bits, usually eight, treated as a unit, and representing a character.

## C

**callable services.** In &cel., a set of services that can be invoked by a COBOL program using the conventional &cel.-defined call interface, and usable by all programs sharing the &cel. conventions.

**called program.** A program that is the object of a CALL statement.

\* **calling program.** A program that executes a CALL to another program.

**case structure.** A program processing logic in which a series of conditions is tested in order to make a choice between a number of resulting actions.

**cataloged procedure.** A set of job control statements placed in a partitioned data set called the procedure library (SYS1.PROCLIB). You can use cataloged procedures to save time and reduce errors coding JCL.

**century window.** The 100-year interval in which Language Environment assumes all 2-digit years lie. The Language Environment default century window begins 80 years before the system date.

\* **character.** The basic indivisible unit of the language.

**character position.** The amount of physical storage required to store a single standard data format character described as USAGE IS DISPLAY.

**character set.** All the valid characters for a programming language or a computer system.

\* **character-string.** A sequence of contiguous characters that form a COBOL word, a literal, a PICTURE character-string, or a comment-entry. Must be delimited by separators.

**checkpoint.** A point at which information about the status of a job and the system can be recorded so that the job step can be later restarted.

\* **class.** The entity that defines common behavior and implementation for zero, one, or more objects. The objects that share the same implementation are considered to be objects of the same class.

\* **class condition.** The proposition, for which a truth value can be determined, that the content of an item is wholly alphabetic, is wholly numeric, or consists exclusively of those characters listed in the definition of a class-name.

\* **Class Definition.** The COBOL source unit that defines a class.

\* **class identification entry.** An entry in the CLASS-ID paragraph of the IDENTIFICATION DIVISION which contains clauses that specify the class-name and assign selected attributes to the class definition.

\* **class-name.** A user-defined word defined in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION that assigns a name to the proposition for which a truth value can be defined, that the content of a data item consists exclusively of those characters listed in the definition of the class-name.

**class object.** The run-time object representing a SOM class.

\* **clause.** An ordered set of consecutive COBOL character-strings whose purpose is to specify an attribute of an entry.

**CMS (Conversational Monitor System).** A virtual machine operating system that provides general interactive, time-sharing, problem solving, and program development capabilities, and that operates only under the control of the VM/SP control program.

\* **COBOL character set.** The complete COBOL character set consists of the characters listed below:

Character	Meaning
0,1...,9	digit
A,B,...,Z	uppercase letter
a,b,...,z	lowercase letter
•	space
+	plus sign
—	minus sign (hyphen)
*	asterisk
/	slant (virgule, slash)
=	equal sign
\$	currency sign
,	comma (decimal point)
;	semicolon
.	period (decimal point, full stop)
”	quotation mark
(	left parenthesis

)	right parenthesis
>	greater than symbol
<	less than symbol
:	colon

\* **COBOL word.** See “word.”

**code page.** An assignment of graphic characters and control function meanings to all code points; for example, assignment of characters and meanings to 256 code points for 8-bit code, assignment of characters and meanings to 128 code points for 7-bit code.

\* **collating sequence.** The sequence in which the characters that are acceptable to a computer are ordered for purposes of sorting, merging, comparing, and for processing indexed files sequentially.

\* **column.** A character position within a print line. The columns are numbered from 1, by 1, starting at the leftmost character position of the print line and extending to the rightmost position of the print line.

\* **combined condition.** A condition that is the result of connecting two or more conditions with the AND or the OR logical operator.

\* **comment-entry.** An entry in the IDENTIFICATION DIVISION that may be any combination of characters from the computer's character set.

\* **comment line.** A source program line represented by an asterisk (\*) in the indicator area of the line and any characters from the computer's character set in area A and area B of that line. The comment line serves only for documentation in a program. A special form of comment line represented by a slant (/) in the indicator area of the line and any characters from the computer's character set in area A and area B of that line causes page ejection prior to printing the comment.

\* **common program.** A program which, despite being directly contained within another program, may be called from any program directly or indirectly contained in that other program.

\* **compile.** (1) To translate a program expressed in a high-level language into a program expressed in an intermediate language, assembly language, or a computer language. (2) To prepare a machine language program from a computer program written in another programming language by making use of the overall logic structure of the program, or generating more than one computer instruction for each symbolic statement, or both, as well as performing the function of an assembler.

\* **compile time.** The time at which a COBOL source program is translated, by a COBOL compiler, to a COBOL object program.

**compiler.** A program that translates a program written in a higher level language into a machine language object program.

**compiler directing statement.** A statement, beginning with a compiler directing verb, that causes the compiler to take a specific action during compilation.

**compiler directing statement.** A statement that specifies actions to be taken by the compiler during processing of a COBOL source program. Compiler directives are contained in the COBOL source program. Thus, you can specify different suboptions of the directive within the source program by using multiple compiler directive statements in the program.

\* **complex condition.** A condition in which one or more logical operators act upon one or more conditions. (See also “negated simple condition,” “combined condition,” and “negated combined condition.”)

\* **computer-name.** A system-name that identifies the computer upon which the program is to be compiled or run.

**condition.** An exception that has been enabled, or recognized, by &cel. and thus is eligible to activate user and language condition handlers. Any alteration to the normal programmed flow of an application. Conditions can be detected by the hardware/operating system and results in an interrupt. They can also be detected by language-specific generated code or language library code.

\* **condition.** A status of a program at run time for which a truth value can be determined. Where the term ‘condition’ (condition-1, condition-2,...) appears in these language specifications in or in reference to ‘condition’ (condition-1, condition-2,...) of a general format, it is a conditional expression consisting of either a simple condition optionally parenthesized, or a combined condition consisting of the syntactically correct combination of simple conditions, logical operators, and parentheses, for which a truth value can be determined.

\* **conditional expression.** A simple condition or a complex condition specified in an EVALUATE, IF, PERFORM, or SEARCH statement. (See also “simple condition” and “complex condition.”)

\* **conditional phrase.** A conditional phrase specifies the action to be taken upon determination of the truth value of a condition resulting from the execution of a conditional statement.

\* **conditional statement.** A statement specifying that the truth value of a condition is to be determined and that the subsequent action of the object program is dependent on this truth value.

\* **conditional variable.** A data item one or more values of which has a condition-name assigned to it.

\* **condition-name.** A user-defined word that assigns a name to a subset of values that a conditional variable may assume; or a user-defined word assigned to a status of an implementor defined switch or device. When ‘condition-name’ is used in the general formats, it represents a unique data item reference consisting of a syntactically correct combination of a ‘condition-name’, together with qualifiers and subscripts, as required for uniqueness of reference.

\* **condition-name condition.** The proposition, for which a truth value can be determined, that the value of a conditional variable is a member of the set of values attributed to a condition-name associated with the conditional variable.

\* **CONFIGURATION SECTION.** A section of the ENVIRONMENT DIVISION that describes overall specifications of source and object programs and class definitions.

**CONSOLE.** A COBOL environment-name associated with the operator console.

\* **contiguous items.** Items that are described by consecutive entries in the Data Division, and that bear a definite hierarchic relationship to each other.

**CORBA.** The Common Object Request Broker Architecture established by the Object Management Group. IBM’s *Interface Definition Language* used to describe the *interface* for SOM classes is fully compliant with CORBA standards.

\* **counter.** A data item used for storing numbers or number representations in a manner that permits these numbers to be increased or decreased by the value of another number, or to be changed or reset to zero or to an arbitrary positive or negative value.

**cross-reference listing.** The portion of the compiler listing that contains information on where files, fields, and indicators are defined, referenced, and modified in a program.

\* **currency sign.** The character ‘\$’ of the COBOL character set or that character defined by the CURRENCY compiler option. If the NOCURRENCY compiler option is in effect, the currency sign is defined as the character ‘\$’.

**currency symbol.** The character defined by the CURRENCY compiler option or by the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph. If the NOCURRENCY compiler option is in effect for a COBOL source program and the CURRENCY SIGN clause is also **not** present in the source program, the currency symbol is identical to the currency sign.

\* **current record.** In file processing, the record that is available in the record area associated with a file.

\* **current volume pointer.** A conceptual entity that points to the current volume of a sequential file.

## D

\* **data clause.** A clause, appearing in a data description entry in the DATA DIVISION of a COBOL program, that provides information describing a particular attribute of a data item.

\* **data description entry .** An entry in the DATA DIVISION of a COBOL program that is composed of a level-number followed by a data-name, if required, and then followed by a set of data clauses, as required.

**DATA DIVISION.** One of the four main components of a COBOL program, class definition, or method definition. The DATA DIVISION describes the data to be processed by the object program, class, or method: files to be used and the records contained within them; internal working-storage records that will be needed; data to be made available in more than one program in the COBOL run unit. (Note, the Class DATA DIVISION contains only the WORKING-STORAGE SECTION.)

\* **data item.** A unit of data (excluding literals) defined by a COBOL program or by the rules for function evaluation.

\* **data-name.** A user-defined word that names a data item described in a data description entry. When used in the general formats, 'data-name' represents a word that must not be reference-modified, subscripted or qualified unless specifically permitted by the rules for the format.

**DBCS (Double-Byte Character Set).** See "Double-Byte Character Set (DBCS)."

\* **debugging line.** A debugging line is any line with a 'D' in the indicator area of the line.

\* **debugging section.** A section that contains a USE FOR DEBUGGING statement.

\* **declarative sentence.** A compiler directing sentence consisting of a single USE statement terminated by the separator period.

\* **declaratives.** A set of one or more special purpose sections, written at the beginning of the Procedure Division, the first of which is preceded by the key word DECLARATIVES and the last of which is followed by the key words END DECLARATIVES. A declarative is composed of a section header, followed by a USE compiler directing sentence, followed by a set of zero, one, or more associated paragraphs.

\* **de-edit.** The logical removal of all editing characters from a numeric edited data item in order to determine that item's unedited numeric value.

\* **delimited scope statement.** Any statement that includes its explicit scope terminator.

\* **delimiter.** A character or a sequence of contiguous characters that identify the end of a string of characters and separate that string of characters from the following string of characters. A delimiter is not part of the string of characters that it delimits.

\* **descending key.** A key upon the values of which data is ordered starting with the highest value of key down to the lowest value of key, in accordance with the rules for comparing data items.

**digit.** Any of the numerals from 0 through 9. In COBOL, the term is not used in reference to any other symbol.

\* **digit position.** The amount of physical storage required to store a single digit. This amount may vary depending on the usage specified in the data description entry that defines the data item.

\* **direct access.** The facility to obtain data from storage devices or to enter data into a storage device in such a way that the process depends only on the location of that data and not on a reference to data previously accessed.

\* **division.** A collection of zero, one or more sections or paragraphs, called the division body, that are formed and combined in accordance with a specific set of rules. Each division consists of the division header and the related division body. There are four (4) divisions in a COBOL program: Identification, Environment, Data, and Procedure.

\* **division header.** A combination of words followed by a separator period that indicates the beginning of a division. The division headers are:

IDENTIFICATION DIVISION.  
ENVIRONMENT DIVISION.  
DATA DIVISION.  
PROCEDURE DIVISION.

**do construction.** In structured programming, a DO statement is used to group a number of statements in a procedure. In COBOL, an in-line PERFORM statement functions in the same way.

**do-until.** In structured programming, a do-until loop will be executed at least once, and until a given condition is true. In COBOL, a TEST AFTER phrase used with the PERFORM statement functions in the same way.

**do-while.** In structured programming, a do-while loop will be executed if, and while, a given condition is

true. In COBOL, a TEST BEFORE phrase used with the PERFORM statement functions in the same way.

**Double-Byte Character Set (DBCS).** A set of characters in which each character is represented by two bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require Double-Byte Character Sets. Because each character requires two bytes, entering, displaying, and printing DBCS characters requires hardware and supporting software that are DBCS-capable.

**\* dynamic access.** An access mode in which specific logical records can be obtained from or placed into a mass storage file in a nonsequential manner and obtained from a file in a sequential manner during the scope of the same OPEN statement.

**Dynamic Storage Area (DSA).** Dynamically acquired storage composed of a register save area and an area available for dynamic storage allocation (such as program variables). DSAs are generally allocated within STACK segments managed by &cel..

## E

**\* EBCDIC (Extended Binary-Coded Decimal Interchange Code).** A coded character set consisting of 8-bit coded characters.

**EBCDIC character.** Any one of the symbols included in the 8-bit EBCDIC (Extended Binary-Coded-Decimal Interchange Code) set.

**edited data item.** A data item that has been modified by suppressing zeroes and/or inserting editing characters.

**\* editing character.** A single character or a fixed two-character combination belonging to the following set:

Character	Meaning
•	space
0	zero
+	plus
—	minus
CR	credit
DB	debit
Z	zero suppress
*	check protect
\$	currency sign
,	comma (decimal point)
.	period (decimal point)
/	slant (virgule, slash)

**element (text element).** One logical unit of a string of text, such as the description of a single data item or verb, preceded by a unique code identifying the element type.

**\* elementary item.** A data item that is described as not being further logically subdivided.

**enclave.** When running under the &cel. product, an enclave is analogous to a run unit. An enclave can create other enclaves on MVS and CMS by a LINK, on CMS by CMSCALL, and the use of the system () function of C.

**\*end class header.** A combination of words, followed by a separator period, that indicates the end of a COBOL class definition. The end class header is:  
END CLASS class-name.

**\*end method header.** A combination of words, followed by a separator period, that indicates the end of a COBOL method definition. The end method header is:

END METHOD method-name.

**\* end of Procedure Division.** The physical position of a COBOL source program after which no further procedures appear.

**\* end program header.** A combination of words, followed by a separator period, that indicates the end of a COBOL source program. The end program header is:

END PROGRAM program-name.

**\* entry.** Any descriptive set of consecutive clauses terminated by a separator period and written in the IDENTIFICATION DIVISION, ENVIRONMENT DIVISION, or DATA DIVISION of a COBOL program.

**\* environment clause.** A clause that appears as part of an ENVIRONMENT DIVISION entry.

**ENVIRONMENT DIVISION.** One of the four main component parts of a COBOL program, class definition, or method definition. The ENVIRONMENT DIVISION describes the computers upon which the source program is compiled and those on which the object program is executed, and provides a linkage between the logical concept of files and their records, and the physical aspects of the devices on which files are stored.

**environment-name.** A name, specified by IBM, that identifies system logical units, printer and card punch control characters, report codes, and/or program switches. When an environment-name is associated with a mnemonic-name in the ENVIRONMENT DIVISION, the mnemonic-name may then be substituted in any format in which such substitution is valid.

**environment variable.** Any of a number of variables that describe the way an operating system is going to run and the devices it is going to recognize.

**execution time.** See “run time.”



**execution-time environment.** See “run-time environment.”

\* **explicit scope terminator.** A reserved word that terminates the scope of a particular Procedure Division statement.

**exponent.** A number, indicating the power to which another number (the base) is to be raised. Positive exponents denote multiplication, negative exponents denote division, fractional exponents denote a root of a quantity. In COBOL, an exponential expression is indicated with the symbol ‘\*\*’ followed by the exponent.

\* **expression.** An arithmetic or conditional expression.

\* **extend mode.** The state of a file after execution of an OPEN statement, with the EXTEND phrase specified for that file, and before the execution of a CLOSE statement, without the REEL or UNIT phrase for that file.

**extensions.** Certain COBOL syntax and semantics supported by IBM compilers in addition to those described in ANSI Standard.

\* **external data.** The data described in a program as external data items and external file connectors.

\* **external data item.** A data item which is described as part of an external record in one or more programs of a run unit and which itself may be referenced from any program in which it is described.

\* **external data record.** A logical record which is described in one or more programs of a run unit and whose constituent data items may be referenced from any program in which they are described.

**external decimal item.** A format for representing numbers in which the digit is contained in bits 4 through 7 and the sign is contained in bits 0 through 3 of the rightmost byte. Bits 0 through 3 of all other bytes contain 1's (hex F). For example, the decimal value of +123 is represented as 1111 0001 1111 0010 1111 0011. (Also known as “zoned decimal item.”)

\* **external file connector.** A file connector which is accessible to one or more object programs in the run unit.

**external floating-point item.** A format for representing numbers in which a real number is represented by a pair of distinct numerals. In a floating-point representation, the real number is the product of the fixed-point part (the first numeral), and a value obtained by raising the implicit floating-point base to a power denoted by the exponent (the second numeral).

For example, a floating-point representation of the number 0.0001234 is: 0.1234 -3, where 0.1234 is the mantissa and -3 is the exponent.

\* **external switch.** A hardware or software device, defined and named by the implementor, which is used to indicate that one of two alternate states exists.

## F

\* **figurative constant.** A compiler-generated value referenced through the use of certain reserved words.

\* **file.** A collection of logical records.

\* **file attribute conflict condition.** An unsuccessful attempt has been made to execute an input-output operation on a file and the file attributes, as specified for that file in the program, do not match the fixed attributes for that file.

\* **file clause.** A clause that appears as part of any of the following DATA DIVISION entries: file description entry (FD entry) and sort-merge file description entry (SD entry).

\* **file connector.** A storage area which contains information about a file and is used as the linkage between a file-name and a physical file and between a file-name and its associated record area.

**File-Control.** The name of an ENVIRONMENT DIVISION paragraph in which the data files for a given source program are declared.

\* **file control entry.** A SELECT clause and all its subordinate clauses which declare the relevant physical attributes of a file.

\* **file description entry.** An entry in the File Section of the DATA DIVISION that is composed of the level indicator FD, followed by a file-name, and then followed by a set of file clauses as required.

\* **file-name.** A user-defined word that names a file connector described in a file description entry or a sort-merge file description entry within the File Section of the DATA DIVISION.

\* **file organization.** The permanent logical file structure established at the time that a file is created.

\* **file position indicator.** A conceptual entity that contains the value of the current key within the key of reference for an indexed file, or the record number of the current record for a sequential file, or the relative record number of the current record for a relative file, or indicates that no next logical record exists, or that an optional input file is not present, or that the at end condition already exists, or that no valid next record has been established.

\* **File Section.** The section of the DATA DIVISION that contains file description entries and sort-merge file description entries together with their associated record descriptions.

**file system.** The collection of files and file management structures on a physical or logical mass storage device, such as a diskette or minidisk.

\* **fixed file attributes.** Information about a file which is established when a file is created and cannot subsequently be changed during the existence of the file. These attributes include the organization of the file (sequential, relative, or indexed), the prime record key, the alternate record keys, the code set, the minimum and maximum record size, the record type (fixed or variable), the collating sequence of the keys for indexed files, the blocking factor, the padding character, and the record delimiter.

\* **fixed length record.** A record associated with a file whose file description or sort-merge description entry requires that all records contain the same number of character positions.

**fixed-point number.** A numeric data item defined with a PICTURE clause that specifies the location of an optional sign, the number of digits it contains, and the location of an optional decimal point. The format may be either binary, packed decimal, or external decimal.

**floating-point number.** A numeric data item containing a fraction and an exponent. Its value is obtained by multiplying the fraction by the base of the numeric data item raised to the power specified by the exponent.

\* **format.** A specific arrangement of a set of data.

\* **function.** A temporary data item whose value is determined at the time the function is referenced during the execution of a statement.

\* **function-identifier.** A syntactically correct combination of character-strings and separators that references a function. The data item represented by a function is uniquely identified by a function-name with its arguments, if any. A function-identifier may include a reference-modifier. A function-identifier that references an alphanumeric function may be specified anywhere in the general formats that an identifier may be specified, subject to certain restrictions. A function-identifier that references an integer or numeric function may be referenced anywhere in the general formats that an arithmetic expression may be specified.

**function-name.** A word that names the mechanism whose invocation, along with required arguments, determines the value of a function.

## G

\* **global name.** A name which is declared in only one program but which may be referenced from that program and from any program contained within that program. Condition-names, data-names, file-names, record-names, report-names, and some special registers may be global names.

\* **group item.** A data item that is composed of subordinate data items.

## H

**header label.** (1) A file label or data set label that precedes the data records on a unit of recording media. (2) Synonym for beginning-of-file label.

\* **high order end.** The leftmost character of a string of characters.

## I

**IBM COBOL extension.** Certain COBOL syntax and semantics supported by IBM compilers in addition to those described in ANSI Standard.

**IDENTIFICATION DIVISION.** One of the four main component parts of a COBOL program, class definition, or method definition. The IDENTIFICATION DIVISION identifies the program name, class name, or method name. The IDENTIFICATION DIVISION may include the following documentation: author name, installation, or date.

\* **identifier.** A syntactically correct combination of character-strings and separators that names a data item. When referencing a data item that is not a function, an identifier consists of a data-name, together with its qualifiers, subscripts, and reference-modifier, as required for uniqueness of reference. When referencing a data item which is a function, a function-identifier is used.

**IGZCBSN.** The &cobol. bootstrap routine. It must be link-edited with any module that contains a &cobol. program.

\* **imperative statement.** A statement that either begins with an imperative verb and specifies an unconditional action to be taken or is a conditional statement that is delimited by its explicit scope terminator (delimited scope statement). An imperative statement may consist of a sequence of imperative statements.

\* **implicit scope terminator.** A separator period which terminates the scope of any preceding unterminated statement, or a phrase of a statement which by its occurrence indicates the end of the scope of any statement contained within the preceding phrase.

\* **index.** A computer storage area or register, the content of which represents the identification of a particular element in a table.

\* **index data item.** A data item in which the values associated with an index-name can be stored in a form specified by the implementor.

**indexed data-name.** An identifier that is composed of a data-name, followed by one or more index-names enclosed in parentheses.

\* **indexed file.** A file with indexed organization.

\* **indexed organization.** The permanent logical file structure in which each record is identified by the value of one or more keys within that record.

**indexing.** Synonymous with subscripting using index-names.

\* **index-name.** A user-defined word that names an index associated with a specific table.

\* **inheritance (for classes).** A mechanism for using the implementation of one or more *classes* as the basis for another class. A *sub-class* inherits from one or more *super-classes*. By definition the inheriting class conforms to the inherited classes.

\* **initial program.** A program that is placed into an initial state every time the program is called in a run unit.

\* **initial state.** The state of a program when it is first called in a run unit.

**inline.** In a program, instructions that are executed sequentially, without branching to routines, subroutines, or other programs.

\* **input file.** A file that is opened in the INPUT mode.

\* **input mode.** The state of a file after execution of an OPEN statement, with the INPUT phrase specified, for that file and before the execution of a CLOSE statement, without the REEL or UNIT phrase for that file.

\* **input-output file.** A file that is opened in the I-O mode.

\* **INPUT-OUTPUT SECTION.** The section of the ENVIRONMENT DIVISION that names the files and the external media required by an object program or method and that provides information required for transmission and handling of data during execution of the object program or method definition.

\* **Input-Output statement.** A statement that causes files to be processed by performing operations upon individual records or upon the file as a unit. The input-output statements are: ACCEPT (with the

identifier phrase), CLOSE, DELETE, DISPLAY, OPEN, READ, REWRITE, SET (with the TO ON or TO OFF phrase), START, and WRITE.

\* **input procedure.** A set of statements, to which control is given during the execution of a SORT statement, for the purpose of controlling the release of specified records to be sorted.

**instance data.** Data defining the state of an object. The instance data introduced by a class is defined in the WORKING-STORAGE SECTION of the DATA DIVISION of the class definition. The state of an object also includes the state of the instance variables introduced by base classes that are inherited by the current class. A separate copy of the instance data is created for each object instance.

\* **integer.** (1) A numeric literal that does not include any digit positions to the right of the decimal point.

(2) A numeric data item defined in the DATA DIVISION that does not include any digit positions to the right of the decimal point.

(3) A numeric function whose definition provides that all digits to the right of the decimal point are zero in the returned value for any possible evaluation of the function.

**integer function.** A function whose category is numeric and whose definition does not include any digit positions to the right of the decimal point.

**interface.** The information that a *client* must know to use a *class*—the names of its *attributes* and the signatures of its *methods*. With direct-to-SOM compilers such as COBOL, the interface to a class may be defined by native language syntax for class definitions. Classes implemented in other languages might have their interfaces defined directly in SOM Interface Definition Language (IDL). The COBOL compiler has a compiler option, IDLGEN, to automatically generate IDL for a COBOL class.

**Interface Definition Language (IDL).** The formal language (independent of any programming language) by which the *interface* for a class of *objects* is defined in a IDL file, which the SOM compiler then interprets to create an implementation template file and binding files. SOM's Interface Definition Language is fully compliant with standards established by the Object Management Group's Common Object Request Broker Architecture (CORBA).

**interlanguage communication (ILC).** The ability of routines written in different programming languages to communicate. ILC support allows the application writer to readily build applications from component routines written in a variety of languages.

**intermediate result.** An intermediate field containing the results of a succession of arithmetic operations.

\* **internal data.** The data described in a program excluding all external data items and external file connectors. Items described in the LINKAGE SECTION of a program are treated as internal data.

\* **internal data item.** A data item which is described in one program in a run unit. An internal data item may have a global name.

**internal decimal item.** A format in which each byte in a field except the rightmost byte represents two numeric digits. The rightmost byte contains one digit and the sign. For example, the decimal value +123 is represented as 0001 0010 0011 1111. (Also known as packed decimal.)

\* **internal file connector.** A file connector which is accessible to only one object program in the run unit.

\* **intra-record data structure.** The entire collection of groups and elementary data items from a logical record which is defined by a contiguous subset of the data description entries which describe that record. These data description entries include all entries whose level-number is greater than the level-number of the first data description entry describing the intra-record data structure.

**intrinsic function.** A pre-defined function, such as a commonly used arithmetic function, called by a built-in function reference.

\* **invalid key condition.** A condition, at object time, caused when a specific value of the key associated with an indexed or relative file is determined to be invalid.

\* **I-O-CONTROL.** The name of an ENVIRONMENT DIVISION paragraph in which object program requirements for rerun points, sharing of same areas by several data files, and multiple file storage on a single input-output device are specified.

\* **I-O-CONTROL entry.** An entry in the I-O-CONTROL paragraph of the ENVIRONMENT DIVISION which contains clauses that provide information required for the transmission and handling of data on named files during the execution of a program.

\* **I-O-Mode.** The state of a file after execution of an OPEN statement, with the I-O phrase specified, for that file and before the execution of a CLOSE statement without the REEL or UNIT phase for that file.

\* **I-O status.** A conceptual entity which contains the two-character value indicating the resulting status of an input-output operation. This value is made available to the program through the use of the FILE STATUS clause in the file control entry for the file.

**iteration structure.** A program processing logic in which a series of statements is repeated while a condition is true or until a condition is true.

## K

**K.** When referring to storage capacity, two to the tenth power; 1024 in decimal notation.

\* **key.** A data item that identifies the location of a record, or a set of data items which serve to identify the ordering of data.

\* **key of reference.** The key, either prime or alternate, currently being used to access records within an indexed file.

\* **key word.** A reserved word or function-name whose presence is required when the format in which the word appears is used in a source program.

**kilobyte (KB).** One kilobyte equals 1024 bytes.

## L

\* **language-name.** A system-name that specifies a particular programming language.

**Language Environment-conforming.** A characteristic of compiler products (&cobol370., &cobol., AD/Cycle C/370, C/C++ for MVS and VM, PL/I for MVS and VM) that produce object code conforming to the Language Environment format.

**last-used state.** A program is in last-used state if its internal values remain the same as when the program was exited (are not reset to their initial values).

\* **letter.** A character belonging to one of the following two sets:

1. Uppercase letters: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z
2. Lowercase letters: a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z

\* **level indicator.** Two alphabetic characters that identify a specific type of file or a position in a hierarchy. The level indicators in the DATA DIVISION are: CD, FD, and SD.

\* **level-number.** A user-defined word, expressed as a two digit number, which indicates the hierarchical position of a data item or the special properties of a data description entry. Level-numbers in the range from 1 through 49 indicate the position of a data item in the hierarchical structure of a logical record. Level-numbers in the range 1 through 9 may be written either as a single digit or as a zero followed by a significant digit. Level-numbers 66, 77 and 88 identify special properties of a data description entry.

\* **library-name.** A user-defined word that names a COBOL library that is to be used by the compiler for a given source program compilation.

\* **library text.** A sequence of text words, comment lines, the separator space, or the separator pseudo-text delimiter in a COBOL library.

**LILIAN DATE.** The number of days since the beginning of the Gregorian calendar. Day one is Friday, October 15, 1582. The Lilian date format is named in honor of Luigi Lilio, the creator of the Gregorian calendar.

\* **LINAGE-COUNTER.** A special register whose value points to the current position within the page body.

**LINKAGE SECTION.** The section in the DATA DIVISION of the called program that describes data items available from the calling program. These data items may be referred to by both the calling and called program.

**literal.** A character-string whose value is specified either by the ordered set of characters comprising the string, or by the use of a figurative constant.

**local.** A set of attributes for a program execution environment indicating culturally sensitive considerations, such as: character code page, collating sequence, date/time format, monetary value representation, numeric value representation, or language.

\* **LOCAL-STORAGE SECTION.** The section of the DATA DIVISION that defines storage that is allocated and freed on a per-invocation basis, depending on the value assigned in their VALUE clauses.

\* **logical operator.** One of the reserved words AND, OR, or NOT. In the formation of a condition, either AND, or OR, or both can be used as logical connectives. NOT can be used for logical negation.

\* **logical record.** The most inclusive data item. The level-number for a record is 01. A record may be either an elementary item or a group of items. The term is synonymous with record.

\* **low order end.** The rightmost character of a string of characters.

## M

**main program.** In a hierarchy of programs and subroutines, the first program to receive control when the programs are run.

\* **mass storage.** A storage medium in which data may be organized and maintained in both a sequential and nonsequential manner.

\* **mass storage device.** A device having a large storage capacity; for example, magnetic disk, magnetic drum.

\* **mass storage file.** A collection of records that is assigned to a mass storage medium.

\* **megabyte (M).** One megabyte equals 1,048,576 bytes.

\* **merge file.** A collection of records to be merged by a MERGE statement. The merge file is created and can be used only by the merge function.

**metaclass.** A SOM class whose instances are SOM class-objects. The methods defined in metaclasses are executed without requiring any object instances of the class to exist, and are frequently used to create instances of the class.

**method.** Procedural code that defines one of the operations supported by an object, and that is executed by an INVOKE statement on that object.

\* **Method Definition.** The COBOL source unit that defines a method.

\* **method identification entry.** An entry in the METHOD-ID paragraph of the IDENTIFICATION DIVISION which contains clauses that specify the method-name and assign selected attributes to the method definition.

\* **method-name.** A user-defined word that identifies a method.

\* **mnemonic-name.** A user-defined word that is associated in the ENVIRONMENT DIVISION with a specified implementor-name.

**multitasking.** Mode of operation that provides for the concurrent, or interleaved, execution of two or more tasks. When running under the &cel. product, multitasking is synonymous with *multithreading*.

**MVS/XA\* (Multiple Virtual Storage/Extended Architecture).** An IBM operating system that manages multiple virtual address spaces in IBM processors operating in extended architecture mode. MVS/XA supports the 31-bit addressing mechanism of extended architecture mode, and therefore, allows an address space as large as 2<sup>31</sup> bytes (2048 megabytes or 2 gigabytes).

## N

**name.** A word composed of not more than 30 characters that defines a COBOL operand.

\* **native character set.** The implementor-defined character set associated with the computer specified in the OBJECT-COMPUTER paragraph.

\* **native collating sequence.** The implementor-defined collating sequence associated with the computer specified in the OBJECT-COMPUTER paragraph.

\* **negated combined condition.** The 'NOT' logical operator immediately followed by a parenthesized combined condition.

\* **negated simple condition.** The 'NOT' logical operator immediately followed by a simple condition.

**nested program.** A program that is directly contained within another program.

\* **next executable sentence.** The next sentence to which control will be transferred after execution of the current statement is complete.

\* **next executable statement.** The next statement to which control will be transferred after execution of the current statement is complete.

\* **next record.** The record that logically follows the current record of a file.

\* **noncontiguous items.** Elementary data items in the WORKING-STORAGE and LINKAGE SECTIONS that bear no hierarchic relationship to other data items.

\* **nonnumeric item.** A data item whose description permits its content to be composed of any combination of characters taken from the computer's character set. Certain categories of nonnumeric items may be formed from more restricted character sets.

\* **nonnumeric literal.** A literal bounded by quotation marks. The string of characters may include any character in the computer's character set.

**null.** Figurative constant used to assign the value of an invalid address to pointer data items. NULLS can be used wherever NULL can be used.

\* **numeric character.** A character that belongs to the following set of digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

**numeric-edited item.** A numeric item that is in such a form that it may be used in printed output. It may consist of external decimal digits from 0 through 9, the decimal point, commas, the dollar sign, editing sign control symbols, plus other editing symbols.

\* **numeric function.** A function whose class and category are numeric but which for some possible evaluation does not satisfy the requirements of integer functions.

\* **numeric item.** A data item whose description restricts its content to a value represented by characters chosen from the digits from '0' through '9'; if signed, the item may also contain a '+', '-', or other representation of an operational sign.

\* **numeric literal.** A literal composed of one or more numeric characters that may contain either a decimal point, or an algebraic sign, or both. The decimal point must not be the rightmost character. The algebraic sign, if present, must be the leftmost character.

## O

**object.** An entity that has state (its data values) and operations (its methods). An object is a way to encapsulate state and behavior.

**object code.** Output from a compiler or assembler that is itself executable machine code or is suitable for processing to produce executable machine code.

\* **OBJECT-COMPUTER.** The name of an ENVIRONMENT DIVISION paragraph in which the computer environment, within which the object program is executed, is described.

\* **object computer entry.** An entry in the OBJECT-COMPUTER paragraph of the ENVIRONMENT DIVISION which contains clauses that describe the computer environment in which the object program is to be executed.

**object deck.** A portion of an object program suitable as input to a linkage editor. Synonymous with *object module* and *text deck*.

**object module.** Synonym for *object deck* or *text deck*.

\* **object of entry.** A set of operands and reserved words, within a DATA DIVISION entry of a COBOL program, that immediately follows the subject of the entry.

\* **object program.** A set or group of executable machine language instructions and other material designed to interact with data to provide problem solutions. In this context, an object program is generally the machine language result of the operation of a COBOL compiler on a source program. Where there is no danger of ambiguity, the word 'program' alone may be used in place of the phrase 'object program.'

\* **object time.** The time at which an object program is executed. The term is synonymous with execution time.

\* **obsolete element.** A COBOL language element in Standard COBOL that is to be deleted from the next revision of Standard COBOL.

**ODO object.** In the example below,

```
WORKING-STORAGE SECTION
01  TABLE-1.
    05  X                                PICS9.
    05  Y OCCURS 3 TIMES
        DEPENDING ON X                PIC X.
```

X is the object of the OCCURS DEPENDING ON clause (ODO object). The value of the ODO object determines how many of the ODO subject appear in the table.

**ODO subject.** In the example above, Y is the subject of the OCCURS DEPENDING ON clause (ODO subject). The number of Y ODO subjects that appear in the table depends on the value of X.

\* **open mode.** The state of a file after execution of an OPEN statement for that file and before the execution of a CLOSE statement without the REEL or UNIT phrase for that file. The particular open mode is specified in the OPEN statement as either INPUT, OUTPUT, I-O or EXTEND.

\* **operand.** Whereas the general definition of operand is "that component which is operated upon," for the purposes of this document, any lowercase word (or words) that appears in a statement or entry format may be considered to be an operand and, as such, is an implied reference to the data indicated by the operand.

\* **operational sign.** An algebraic sign, associated with a numeric data item or a numeric literal, to indicate whether its value is positive or negative.

\* **optional file.** A file which is declared as being not necessarily present each time the object program is executed. The object program causes an interrogation for the presence or absence of the file.

\* **optional word.** A reserved word that is included in a specific format only to improve the readability of the language and whose presence is optional to the user when the format in which the word appears is used in a source program.

**OS/2 (Operating System/2\*).** A multi-tasking operating system for the IBM Personal Computer family that allows you to run both DOS mode and OS/2 mode programs.

\* **output file.** A file that is opened in either the OUTPUT mode or EXTEND mode.

\* **output mode.** The state of a file after execution of an OPEN statement, with the OUTPUT or EXTEND phrase specified, for that file and before the execution of a CLOSE statement without the REEL or UNIT phrase for that file.

\* **output procedure.** A set of statements to which control is given during execution of a SORT statement after the sort function is completed, or during execution of a MERGE statement after the merge function reaches a point at which it can select the next record in merged order when requested.

**overflow condition.** A condition that occurs when a portion of the result of an operation exceeds the capacity of the intended unit of storage.

## P

**packed decimal item.** See "internal decimal item."

\* **padding character.** An alphanumeric character used to fill the unused character positions in a physical record.

**page.** A vertical division of output data representing a physical separation of such data, the separation being based on internal logical requirements and/or external characteristics of the output medium.

\* **page body.** That part of the logical page in which lines can be written and/or spaced.

\* **paragraph.** In the Procedure Division, a paragraph-name followed by a separator period and by zero, one, or more sentences. In the IDENTIFICATION and ENVIRONMENT DIVISIONs, a paragraph header followed by zero, one, or more entries.

\* **paragraph header.** A reserved word, followed by the separator period, that indicates the beginning of a paragraph in the IDENTIFICATION and ENVIRONMENT DIVISIONs. The permissible paragraph headers in the IDENTIFICATION DIVISION are:

PROGRAM-ID. (Program IDENTIFICATION DIVISION only)  
CLASS-ID. (Class IDENTIFICATION DIVISION only)  
METHOD-ID. (Method IDENTIFICATION DIVISION only)  
AUTHOR.  
INSTALLATION.  
DATE-WRITTEN.  
DATE-COMPILED.  
SECURITY.

The permissible paragraph headers in the ENVIRONMENT DIVISION are:

SOURCE-COMPUTER.  
OBJECT-COMPUTER.  
SPECIAL-NAMES.  
REPOSITORY. (Program or Class CONFIGURATION SECTION only)  
FILE-CONTROL.  
I-O-CONTROL.

\* **paragraph-name.** A user-defined word that identifies and begins a paragraph in the Procedure Division.

**parameter.** Parameters are used to pass data values between calling and called programs.

**password.** A unique string of characters that a program, computer operator, or user must supply to meet security requirements before gaining access to data.

\* **phrase.** A phrase is an ordered set of one or more consecutive COBOL character-strings that form a portion of a COBOL procedural statement or of a COBOL clause.

\* **physical record.** See "block."

**pointer data item.** A data item in which address values can be stored. Data items are explicitly defined as pointers with the USAGE IS POINTER clause. ADDRESS OF special registers are implicitly defined as pointer data items. Pointer data items can be compared for equality or moved to other pointer data items.

**portability.** The ability to transfer an application program from one application platform to another with relatively few changes to the source program.

**preloaded.** In COBOL this refers to COBOL programs that remain resident in storage under IMS instead of being loaded each time they are called.

\* **prime record key.** A key whose contents uniquely identify a record within an indexed file.

\* **priority-number.** A user-defined word which classifies sections in the Procedure Division for purposes of segmentation. Segment-numbers may contain only the characters '0','1', ... , '9'. A segment-number may be expressed either as a one- or two-digit number.

\* **procedure.** A paragraph or group of logically successive paragraphs, or a section or group of logically successive sections, within the Procedure Division.

\* **procedure branching statement.** A statement that causes the explicit transfer of control to a statement other than the next executable statement in the sequence in which the statements are written in the source program. The procedure branching statements are: ALTER, CALL, EXIT, EXIT PROGRAM, GO TO, MERGE, (with the OUTPUT PROCEDURE phrase), PERFORM and SORT (with the INPUT PROCEDURE or OUTPUT PROCEDURE phrase).

**Procedure Division.** One of the four main component parts of a COBOL program, class definition, or method definition. The Procedure Division contains instructions for solving a problem. The Program and Method Procedure Divisions may contain imperative statements, conditional statements, compiler directing statements, paragraphs, procedures, and sections. The Class Procedure Division contains only method definitions.

**procedure integration.** One of the functions of the COBOL optimizer is to simplify calls to performed procedures or contained programs.

PERFORM procedure integration is the process whereby a PERFORM statement is replaced by its

performed procedures. Contained program procedure integration is the process where a CALL to a contained program is replaced by the program code.

\* **procedure-name.** A user-defined word that is used to name a paragraph or section in the Procedure Division. It consists of a paragraph-name (which may be qualified) or a section-name.

**procedure-pointer data item.** A data item in which a pointer to an entry point can be stored. A data item defined with the USAGE IS PROCEDURE-POINTER clause contains the address of a procedure entry point.

\* **program identification entry.** An entry in the PROGRAM-ID paragraph of the IDENTIFICATION DIVISION which contains clauses that specify the program-name and assign selected program attributes to the program.

\* **program-name.** In the IDENTIFICATION DIVISION and the end program header, a user-defined word that identifies a COBOL source program.

\* **pseudo-text.** A sequence of text words, comment lines, or the separator space in a source program or COBOL library bounded by, but not including, pseudo-text delimiters.

\* **pseudo-text delimiter.** Two contiguous equal sign characters (==) used to delimit pseudo-text.

\* **punctuation character.** A character that belongs to the following set:

Character	Meaning
,	comma
;	semicolon
:	colon
.	period (full stop)
"	quotation mark
(	left parenthesis
)	right parenthesis
•	space
=	equal sign

## Q

**QSAM (Queued Sequential Access Method).** An extended version of the basic sequential access method (BSAM). When this method is used, a queue is formed of input data blocks that are awaiting processing or of output data blocks that have been processed and are awaiting transfer to auxiliary storage or to an output device.

\* **qualified data-name.** An identifier that is composed of a data-name followed by one or more sets of either of the connectives OF and IN followed by a data-name qualifier.

\* **qualifier.**



1. A data-name or a name associated with a level indicator which is used in a reference either together with another data-name which is the name of an item that is subordinate to the qualifier or together with a condition-name.
2. A section-name that is used in a reference together with a paragraph-name specified in that section.
3. A library-name that is used in a reference together with a text-name associated with that library.

## R

\* **random access.** An access mode in which the program-specified value of a key data item identifies the logical record that is obtained from, deleted from, or placed into a relative or indexed file.

\* **record.** See "logical record."

\* **record area.** A storage area allocated for the purpose of processing the record described in a record description entry in the File Section of the DATA DIVISION. In the File Section, the current number of character positions in the record area is determined by the explicit or implicit RECORD clause.

\* **record description.** See "record description entry."

\* **record description entry.** The total set of data description entries associated with a particular record. The term is synonymous with record description.

**recording mode.** The format of the logical records in a file. Recording mode can be F (fixed-length), V (variable-length), S (spanned), or U (undefined).

**record key.** A key whose contents identify a record within an indexed file.

\* **record-name.** A user-defined word that names a record described in a record description entry in the DATA DIVISION of a COBOL program.

\* **record number.** The ordinal number of a record in the file whose organization is sequential.

**recursion.** A program calling itself or being directly or indirectly called by a one of its called programs.

**recursively capable.** A program is recursively capable (can be called recursively) if the RECURSIVE attribute is on the PROGRAM-ID statement.

**reel.** A discrete portion of a storage medium, the dimensions of which are determined by each implementor that contains part of a file, all of a file, or any number of files. The term is synonymous with unit and volume.

**reentrant.** The attribute of a program or routine that allows more than one user to share a single copy of a load module.

\* **reference format.** A format that provides a standard method for describing COBOL source programs.

**reference modification.** A method of defining a new alphanumeric data item by specifying the leftmost character and length relative to the leftmost character of another alphanumeric data item.

\* **reference-modifier.** A syntactically correct combination of character-strings and separators that defines a unique data item. It includes a delimiting left parenthesis separator, the leftmost character position, a colon separator, optionally a length, and a delimiting right parenthesis separator.

\* **relation.** See "relational operator" or "relation condition."

\* **relational operator.** A reserved word, a relation character, a group of consecutive reserved words, or a group of consecutive reserved words and relation characters used in the construction of a relation condition. The permissible operators and their meanings are:

Operator	Meaning
IS GREATER THAN	Greater than
IS >	Greater than
IS NOT GREATER THAN	Not greater than
IS NOT >	Not greater than
IS LESS THAN	Less than
IS <	Less than
IS NOT LESS THAN	Not less than
IS NOT <	Not less than
IS EQUAL TO	Equal to
IS =	Equal to
IS NOT EQUAL TO	Not equal to
IS NOT =	Not equal to
IS GREATER THAN OR EQUAL TO	Greater than or equal to
IS >=	Greater than or equal to
IS LESS THAN OR EQUAL TO	Less than or equal to
IS <=	Less than or equal to

\* **relation character.** A character that belongs to the following set:

Character	Meaning
>	greater than
<	less than
=	equal to

\* **relation condition.** The proposition, for which a truth value can be determined, that the value of an arithmetic expression, data item, nonnumeric literal, or index-name has a specific relationship to the value of another arithmetic expression, data item, nonnumeric literal, or index name. (See also "relational operator.")

\* **relative file.** A file with relative organization.

\* **relative key.** A key whose contents identify a logical record in a relative file.

\* **relative organization.** The permanent logical file structure in which each record is uniquely identified by an integer value greater than zero, which specifies the record's logical ordinal position in the file.

\* **relative record number.** The ordinal number of a record in a file whose organization is relative. This number is treated as a numeric literal which is an integer.

\* **reserved word.** A COBOL word specified in the list of words that may be used in a COBOL source program, but that must not appear in the program as user-defined words or system-names.

\* **resource.** A facility or service, controlled by the operating system, that can be used by an executing program.

\* **resultant identifier.** A user-defined data item that is to contain the result of an arithmetic operation.

**reusable environment.** A reusable environment is when you establish an assembler program as the main program by using either ILBOSTP0 programs, IGZERRE programs, or the RTEREUS run-time option.

**routine.** A set of statements in a COBOL program that causes the computer to perform an operation or series of related operations. In &cel., refers to either a procedure, function, or subroutine.

\* **routine-name.** A user-defined word that identifies a procedure written in a language other than COBOL.

\* **run time.** The time at which an object program is executed. The term is synonymous with object time.

**run-time environment.** The environment in which a COBOL program executes.

\* **run unit.** A stand-alone object program, or several object programs, that interact via COBOL CALL statements, which function at run time as an entity.

## S

**SBCS (Single Byte Character Set).** See "Single Byte Character Set (SBCS)".

**scope terminator.** A COBOL reserved word that marks the end of certain Procedure Division statements. It may be either explicit (END-ADD, for example) or implicit (separator period).

\* **section.** A set of zero, one or more paragraphs or entities, called a section body, the first of which is preceded by a section header. Each section consists of the section header and the related section body.

\* **section header.** A combination of words followed by a separator period that indicates the beginning of a section in the Environment, Data, and Procedure Divisions. In the ENVIRONMENT and DATA DIVISIONs, a section header is composed of reserved words followed by a separator period. The permissible section headers in the ENVIRONMENT DIVISION are:

CONFIGURATION SECTION.  
INPUT-OUTPUT SECTION.

The permissible section headers in the DATA DIVISION are:

FILE SECTION.  
WORKING-STORAGE SECTION.  
LOCAL-STORAGE SECTION.  
LINKAGE SECTION.

In the Procedure Division, a section header is composed of a section-name, followed by the reserved word SECTION, followed by a separator period.

\* **section-name.** A user-defined word that names a section in the Procedure Division.

**selection structure.** A program processing logic in which one or another series of statements is executed, depending on whether a condition is true or false.

\* **sentence.** A sequence of one or more statements, the last of which is terminated by a separator period.

\* **separately compiled program.** A program which, together with its contained programs, is compiled separately from all other programs.

\* **separator.** A character or two contiguous characters used to delimit character-strings.

\* **separator comma.** A comma (,) followed by a space used to delimit character-strings.

\* **separator period.** A period (.) followed by a space used to delimit character-strings.

\* **separator semicolon.** A semicolon (;) followed by a space used to delimit character-strings.

**sequence structure.** A program processing logic in which a series of statements is executed in sequential order.

\* **sequential access.** An access mode in which logical records are obtained from or placed into a file in a consecutive predecessor-to-successor logical record sequence determined by the order of records in the file.

\* **sequential file.** A file with sequential organization.

\* **sequential organization.** The permanent logical file structure in which a record is identified by a predecessor-successor relationship established when the record is placed into the file.

**serial search.** A search in which the members of a set are consecutively examined, beginning with the first member and ending with the last.

\* **77-level-description-entry.** A data description entry that describes a noncontiguous data item with the level-number 77.

\* **sign condition.** The proposition, for which a truth value can be determined, that the algebraic value of a data item or an arithmetic expression is either less than, greater than, or equal to zero.

\* **simple condition.** Any single condition chosen from the set:

- Relation condition
- Class condition
- Condition-name condition
- Switch-status condition
- Sign condition

**Single Byte Character Set (SBCS).** A set of characters in which each character is represented by a single byte. See also "EBCDIC (Extended Binary-Coded Decimal Interchange Code)."

**slack bytes.** Bytes inserted between data items or records to ensure correct alignment of some numeric items. Slack bytes contain no meaningful data. In some cases, they are inserted by the compiler; in others, it is the responsibility of the programmer to insert them. The SYNCHRONIZED clause instructs the compiler to insert slack bytes when they are needed for proper alignment. Slack bytes between records are inserted by the programmer.

**SOM.** *System Object Model*

\* **sort file.** A collection of records to be sorted by a SORT statement. The sort file is created and can be used by the sort function only.

\* **sort-merge file description entry.** An entry in the File Section of the DATA DIVISION that is composed of the level indicator SD, followed by a file-name, and then followed by a set of file clauses as required.

\* **SOURCE-COMPUTER.** The name of an ENVIRONMENT DIVISION paragraph in which the computer environment, within which the source program is compiled, is described.

\* **source computer entry.** An entry in the SOURCE-COMPUTER paragraph of the ENVIRONMENT DIVISION which contains clauses that describe the computer environment in which the source program is to be compiled.

\* **source item.** An identifier designated by a SOURCE clause that provides the value of a printable item.

**source program.** Although it is recognized that a source program may be represented by other forms and symbols, in this document it always refers to a syntactically correct set of COBOL statements. A COBOL source program commences with the IDENTIFICATION DIVISION or a COPY statement. A COBOL source program is terminated by the end program header, if specified, or by the absence of additional source program lines.

\* **special character.** A character that belongs to the following set:

Character	Meaning
+	plus sign
-	minus sign (hyphen)
*	asterisk
/	slant (virgule, slash)
=	equal sign
\$	currency sign
,	comma (decimal point)
;	semicolon
.	period (decimal point, full stop)
"	quotation mark
(	left parenthesis
)	right parenthesis
>	greater than symbol
<	less than symbol
:	colon

\* **special-character word.** A reserved word that is an arithmetic operator or a relation character.

**SPECIAL-NAMES.** The name of an ENVIRONMENT DIVISION paragraph in which environment-names are related to user-specified mnemonic-names.

\* **special names entry.** An entry in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION which provides means for specifying the currency sign; choosing the decimal point; specifying symbolic characters; relating implementor-names to user-specified mnemonic-names; relating alphabet-names to character sets or collating

sequences; and relating class-names to sets of characters.

\* **special registers.** Certain compiler generated storage areas whose primary use is to store information produced in conjunction with the use of a specific COBOL feature.

\* **standard data format.** The concept used in describing the characteristics of data in a COBOL DATA DIVISION under which the characteristics or properties of the data are expressed in a form oriented to the appearance of the data on a printed page of infinite length and breadth, rather than a form oriented to the manner in which the data is stored internally in the computer, or on a particular external medium.

\* **statement.** A syntactically valid combination of words, literals, and separators, beginning with a verb, written in a COBOL source program.

**structured programming.** A technique for organizing and coding a computer program in which the program comprises a hierarchy of segments, each segment having a single entry point and a single exit point. Control is passed downward through the structure without unconditional branches to higher levels of the hierarchy.

\* **sub-class.** A class that inherits from another class. When two classes in an inheritance relationship are considered together, the sub-class is the inheritor or inheriting class; the *super-class* is the inheritee or inherited class.

\* **subject of entry.** An operand or reserved word that appears immediately following the level indicator or the level-number in a DATA DIVISION entry.

\* **subprogram.** See "called program."

\* **subscript.** An occurrence number represented by either an integer, a data-name optionally followed by an integer with the operator + or -, or an index-name optionally followed by an integer with the operator + or -, that identifies a particular element in a table. A subscript may be the word ALL when the subscripted identifier is used as a function argument for a function allowing a variable number of arguments.

\* **subscripted data-name.** An identifier that is composed of a data-name followed by one or more subscripts enclosed in parentheses.

\* **super-class.** A class that is inherited by another class. See also *sub-class*.

**switch-status condition.** The proposition, for which a truth value can be determined, that an UPSI switch, capable of being set to an 'on' or 'off' status, has been set to a specific status.

\* **symbolic-character.** A user-defined word that specifies a user-defined figurative constant.

**syntax.** (1) The relationship among characters or groups of characters, independent of their meanings or the manner of their interpretation and use. (2) The structure of expressions in a language. (3) The rules governing the structure of a language. (4) The relationship among symbols. (5) The rules for the construction of a statement.

\* **system-name.** A COBOL word that is used to communicate with the operating environment.

**System Object Model (SOM).** IBM's object-oriented programming technology for building, packaging, and manipulating class libraries. SOM conforms to the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) standards.

## T

\* **table.** A set of logically consecutive items of data that are defined in the DATA DIVISION by means of the OCCURS clause.

\* **table element.** A data item that belongs to the set of repeated items comprising a table.

**text deck.** Synonym for *object deck* or *object module*.

\* **text-name.** A user-defined word that identifies library text.

\* **text word.** A character or a sequence of contiguous characters between margin A and margin R in a COBOL library, source program, or in pseudo-text which is:

- A separator, except for: space; a pseudo-text delimiter; and the opening and closing delimiters for nonnumeric literals. The right parenthesis and left parenthesis characters, regardless of context within the library, source program, or pseudo-text, are always considered text words.
- A literal including, in the case of nonnumeric literals, the opening quotation mark and the closing quotation mark that bound the literal.
- Any other sequence of contiguous COBOL characters except comment lines and the word 'COPY' bounded by separators that are neither a separator nor a literal.

**top-down design.** The design of a computer program using a hierarchic structure in which related functions are performed at each level of the structure.

**top-down development.** See "structured programming."

**trailer-label.** (1) A file or data set label that follows the data records on a unit of recording medium. (2) Synonym for end-of-file label.

\* **truth value.** The representation of the result of the evaluation of a condition in terms of one of two values: true or false.

## U

\* **unary operator.** A plus (+) or a minus (-) sign, that precedes a variable or a left parenthesis in an arithmetic expression and that has the effect of multiplying the expression by +1 or -1, respectively.

**unit.** A module of direct access, the dimensions of which are determined by IBM.

**universal object reference.** A data-name that can refer to an object of any class.

\* **unsuccessful execution.** The attempted execution of a statement that does not result in the execution of all the operations specified by that statement. The unsuccessful execution of a statement does not affect any data referenced by that statement, but may affect status indicators.

**UPSI switch.** A program switch that performs the functions of a hardware switch. Eight are provided: UPSI-0 through UPSI-7.

\* **user-defined word.** A COBOL word that must be supplied by the user to satisfy the format of a clause or statement.

## V

\* **variable.** A data item whose value may be changed by execution of the object program. A variable used in an arithmetic expression must be a numeric elementary item.

\* **variable length record.** A record associated with a file whose file description or sort-merge description entry permits records to contain a varying number of character positions.

\* **variable occurrence data item.** A variable occurrence data item is a table element which is repeated a variable number of times. Such an item must contain an OCCURS DEPENDING ON clause in its data description entry, or be subordinate to such an item.

\* **variably located group.** A group item following, and not subordinate to, a variable-length table in the same level-01 record.

\* **variably located item.** A data item following, and not subordinate to, a variable-length table in the same level-01 record.

\* **verb.** A word that expresses an action to be taken by a COBOL compiler or object program.

**VM/SP (Virtual Machine/System Product).** An IBM-licensed program that manages the resources of a single computer so that multiple computing systems appear to exist. Each virtual machine is the functional equivalent of a "real" machine.

**volume.** A module of external storage. For tape devices it is a reel; for direct-access devices it is a unit.

**volume switch procedures.** System specific procedures executed automatically when the end of a unit or reel has been reached before end-of-file has been reached.

## W

\* **word.** A character-string of not more than 30 characters which forms a user-defined word, a system-name, a reserved word, or a function-name.

\* **WORKING-STORAGE SECTION.** The section of the DATA DIVISION that describes working storage data items, composed either of noncontiguous items or working storage records or of both.

## Z

**zoned decimal item.** See "external decimal item."



---

## List of Abbreviations

<b>AD</b>	application development	<b>FAT</b>	file allocation table GUI
<b>API</b>	application program interface		graphical user interface
<b>CORBA</b>	Common Object Request Broker	<b>IBM</b>	International Business Machines Corporation
<b>CRC</b>	class responsibilities collaborations	<b>IT</b>	information technology
<b>DDE</b>	dynamic data exchange	<b>ITSO</b>	International Technical Support Organization
<b>DDL</b>	data description language	<b>LE</b>	language environment
<b>DFD</b>	data flow diagram	<b>MVC</b>	model view controller
<b>DLL</b>	dynamic link library	<b>OMG</b>	Object Management Group
<b>DSOM</b>	distributed system object model	<b>OTC</b>	Object Technology Council
<b>ER</b>	entity relationship	<b>PAM</b>	project access method
		<b>PDS</b>	partitioned data set
		<b>SOM</b>	system object model





---

## Index

### A

analyzing objects 114  
ANSI standards 35

### C

class 16, 59, 60, 65  
class instance 16  
class programs 56  
class structure  
    as objects 59  
    COBOL program structure 61  
    definition statements 62  
client programs 56  
client structure  
    COBOL program structure 69  
    definition statements  
    example logic flow 69  
client/class relationship 32  
compiler command 56  
CORBA (Common Object Request Broker  
    Architecture) 41  
CRC cards 113

### D

DSOM (Distributed System Object Model) 40

### E

encapsulation 12, 13, 19, 26  
explicit definitions 31

### I

inheritance 12, 14, 19, 26

### L

limitations of 3GL COBOL 28

### M

message 17  
metaclass structure  
    definition statements 99  
metaclasses 97  
method structure  
    COBOL program structure 65  
    definition statements 66  
methods 13, 59, 60, 65

### O

object interaction diagram 138, 158, 176  
object language comparison 43  
object references 71  
objects definition 59  
OMG (Object Management Group) 8, 22, 40  
OO analysis and design 113  
OO analysis procedures  
    CRC card formulation 113  
    method determination 113  
    object identification 113  
    use cases 113  
OO COBOL implementation 47  
OO design 114  
OTC (Object Technology Council) 8

### P

polymorphism 12, 15, 19, 27

### R

recursion 33

### S

SOM (System Object Model) 5, 37, 61, 62, 121, 123  
subclass structure  
    definition statements 83  
    method definition 84  
subclasses 81  
superclass 83  
superclass data acquisition 85

### U

use case analysis  
    basic 114  
    class coding 121  
    coding process 121  
    methods 116  
    objects 115  
    testing client 121

### W

wrapping 47



**International Technical Support Organization  
IBM VisualAge for COBOL for OS/2  
Object-Oriented Programming  
January 1996**

**Publication No. SG24-4606-00**

Your feedback is very important to help us maintain the quality of ITSO Bulletins. **Please fill out this questionnaire and return it using one of the following methods:**

- Mail it to the address on the back (postage paid in U.S. only)
- Give it to an IBM marketing representative for mailing
- Fax it to: Your International Access Code + 1 914 432 8246
- Send a note to REDBOOK@VNET.IBM.COM

**Please rate on a scale of 1 to 5 the subjects below.**

**(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)**

**Overall Satisfaction** \_\_\_\_\_

Organization of the book \_\_\_\_\_  
Accuracy of the information \_\_\_\_\_  
Relevance of the information \_\_\_\_\_  
Completeness of the information \_\_\_\_\_  
Value of illustrations \_\_\_\_\_

Grammar/punctuation/spelling \_\_\_\_\_  
Ease of reading and understanding \_\_\_\_\_  
Ease of finding information \_\_\_\_\_  
Level of technical detail \_\_\_\_\_  
Print quality \_\_\_\_\_

**Please answer the following questions:**

- a) If you are an employee of IBM or its subsidiaries:
- Do you provide billable services for 20% or more of your time? Yes\_\_\_\_\_ No\_\_\_\_\_
- Are you in a Services Organization? Yes\_\_\_\_\_ No\_\_\_\_\_
- b) Are you working in the USA? Yes\_\_\_\_\_ No\_\_\_\_\_
- c) Was the Bulletin published in time for your needs? Yes\_\_\_\_\_ No\_\_\_\_\_
- d) Did this Bulletin meet your needs? Yes\_\_\_\_\_ No\_\_\_\_\_

If no, please explain:

\_\_\_\_\_  
\_\_\_\_\_

What other topics would you like to see in this Bulletin?

\_\_\_\_\_  
\_\_\_\_\_

What other Technical Bulletins would you like to see published?

\_\_\_\_\_

**Comments/Suggestions: ( THANK YOU FOR YOUR FEEDBACK! )**

\_\_\_\_\_  
Name

\_\_\_\_\_  
Address

\_\_\_\_\_  
Company or Organization

\_\_\_\_\_  
Phone No.



Fold and Tape

Please do not staple

Fold and Tape



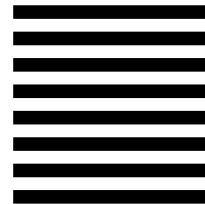
## BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM International Technical Support Organization  
Department 471/E2  
650 Harry Road  
San Jose, CA  
USA 95120-6099

NO POSTAGE  
NECESSARY  
IF MAILED IN THE  
UNITED STATES



Fold and Tape

Please do not staple

Fold and Tape





Printed in U.S.A.

S624-4606-00

